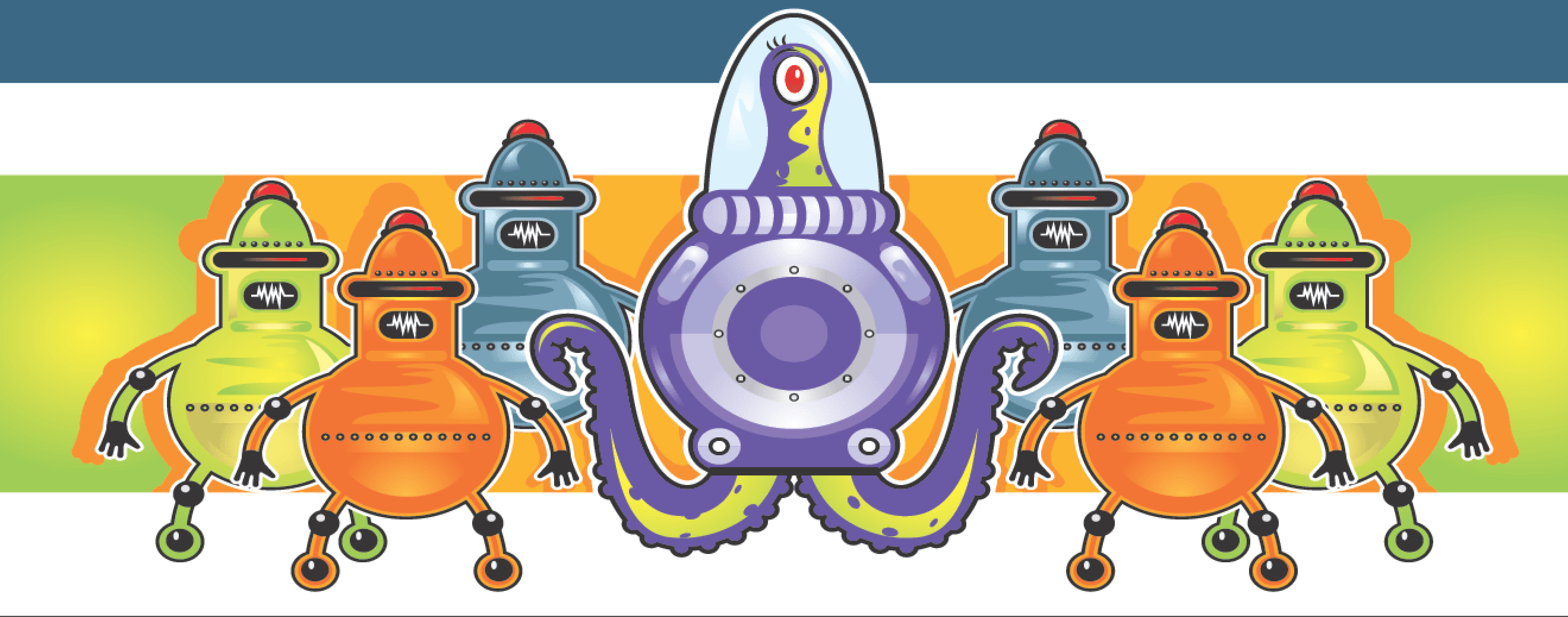
Python 3.5.x、Python 3.6.x

及更新版本的内置对象和标准库对象的高级用法，注重Python内功修炼



Python 程序设计开发宝典

董付国◎著

◇ 150个案例源代码和超过1000个演示性代码片段，完美诠释Pythonic真谛。

◇ 实时跟踪Python动态，紧跟时代发展潮流，介绍GPU加速、分布式、异步通信等最新技术特性。

清华大学出版社

Python 程序设计开发宝典

董付国　著

清 华 大 学 出 版 社

北　　京

内 容 简 介

全书共13章，面向 Python 3.5.x、Python 3.6.x 和 Python 3.7.x，重点关注 Python 内置对象和标准库对象的高级应用以及比较前沿的一些新技术，偶尔涉及一点扩展库用法。第1章介绍 Python 语言编程规范与代码优化建议、开发环境配置、程序伪编译与打包。第2章详解运算符与内置函数的用法。第3章详解各种序列对象、推导式、切片和序列解包。第4章详解选择结构和循环结构，关键字 else、break 与 continue。第5章详解函数的基本用法，可调用对象与修饰器、函数参数、变量的作用域、生成器函数、lambda 表达式、函数柯里化、泛型函数、协程函数和回调函数。第6章详解类的定义、不同类型的成员、依赖注入技术和运算符重载。第7章详解字符串编码与格式化方法、字符串对象方法、文本排版与压缩、汉字拼音有关的技术。第8章详解正则表达式语法、re 模块、正则表达式对象与 match 对象。第9章详解文件对象用法、文件内容操作。第10章详解文件与文件夹操作。第11章详解异常处理结构、文档测试与单元测试、覆盖测试与软件性能测试、代码调试技术。第12章详解不同类型的并行处理技术。第13章详解 asyncio 提供的网络通信功能。

本书不但可以作为 Python 程序设计教材，还可作为 Python 开发工程师的指导用书。

图书在版编目(CIP)数据

Python 程序设计开发宝典/董付国著. —北京：清华大学出版社，2017(2018.1 重印)
ISBN 978-7-302-47210-0

Ⅰ.①P…　Ⅱ.①董…　Ⅲ.①软件工具—程序设计　Ⅳ.①TP311.561

中国版本图书馆 CIP 数据核字(2017)第125790号

责任编辑：白立军
封面设计：杨玉兰
责任校对：焦丽丽
责任印制：王静怡

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编**：100084
社 总 机：010-62770175　　**邮　　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
课件下载：http://www.tup.com.cn，010-62795954
印 装 者：三河市金元印装有限公司
经　　销：全国新华书店
开　　本：185mm×260mm　　**印　张**：25.25　　**字　　数**：584千字
版　　次：2017年8月第1版　　**印　　次**：2018年1月第3次印刷
印　　数：4001～5500
定　　价：69.00元

产品编号：072406-01

前 言



FOREWORD

作者第一次接触 Python 大概是在 2002 年，在几个著名的安全网站上看到有人用这个语言，当时的作者正痴迷于 C 语言和汇编语言，内心不屑于学习这种脚本语言。2010 年的时候因为要用 IDA Pro 分析一个 PE 文件而不得不借助于 IDAPython 插件，才真正感觉到了 Python 的方便与强大，于是购买了很多书籍并且阅读了大量在线文档开始系统地学习 Python。2013 年学校组织各专业教研室修订新版人才培养方案时，经过慎重考虑并且与其他几位老师进行了多次沟通，最终确定为数字媒体技术专业和服务外包专业增加了“Python 程序设计”这门课程。然而，虽然当时市面上已经有了一些 Python 书籍，但是适合作为教材的却寥寥无几，有的过于专注某个专业领域，有的则是泛泛地介绍一点皮毛，并且很多书里放置的插图占用了大量篇幅，知识密度很小，不是作者喜欢的风格。在翻看了超过 20 本 Python 图书以后，作者决定动手写一本适合作教材的书，于是就有了面向计算机及相关专业的《Python 程序设计》（书号为 9787302407232，2015 年 8 月出版）和面向非计算机专业的《Python 程序设计基础》（书号为 9787302410584，2015 年 8 月出版，2016 年 3 月第 2 次印刷，2017 年 1 月第 3 次印刷），前者主要介绍 Python 2.7.x 的基本语法以及在各领域的应用，后者主要介绍 Python 3.4.x 的基本语法而没有涉及太多的应用。本来当时写这两本书的目的仅仅是为了自己上课用起来方便，然而出版不到一年就被近 30 所院校选作教材，还有几十所院校的图书馆也采购了这两本书供学生借阅，反响非常好。应广大读者和用书老师的要求，也为了紧跟 Python 飞速发展的步伐，作者于 2016 年 6 月又出版了《Python 程序设计（第 2 版）》（书号为 9787302436515），这本书使用 Python 3.5.x 重写了第 1 版中的所有案例，并且新增案例近百个，出版后迅速被多所院校选作教材，各大网上书店也频频缺货，2016 年 11 月初进行了第 2 次印刷，2017 年 3 月进行了第 3 次印刷。考虑到更多的 Python 爱好者并没有听老师讲课的机会，看教材自学可能比较吃力，作者于 2017 年 1 月份出版了《Python 可以这样学》（书号为 9787302456469），全书 500 多页，使用生动活泼的语言讲解 Python 3.5.x 的知识和应用（绝大部分内容也适用于 Python 3.6.x 和 Python 3.7.x），在《Python 程序设计（第 2 版）》的基础上删掉了“软件分析与逆向工程”和“安卓平台的 Python 编程”内容，新增了大量案例，并且融入了道德经、周易、太极拳理论中的核心思想以及大量中外名人名言，通过小提示、小技巧、注意、拓展知识等多种形式扩充了大量知识，尤其适合 Python 爱好者自学，也可作为进阶工具书进行查阅。该书出版之后迅速得到社会各界人士的一致认可，第一批印刷的书很快被抢购一空，不到两个月就进行了第二次印刷。回头想想，自己二十年如一日地每天熬夜看书学习写代码还是值得的。

当 2016 年 6 月份作者前几本 Python 系列图书的责任编辑白立军老师约作者再编写一本面向高级程序员的 Python 图书时，说实话，内心是喜悦的，很高兴自己的努力得到广大 Python 爱好者的认可。但同时作者内心也有些担心，感觉在写前 4 本书时已经用完了自己的洪荒之力。在这本新书里再写点什么好呢？内容该如何组织呢？如何避免过多地重复利用前几本书里的案例呢？反复思考了近 2 个月，考虑目前很多 Python 程序员的现状：喜欢直接使用各种扩展库来解决问题，不重视对 Python 语言本身的理解，内功不够深厚，导致很多代码粗制滥造。最终作者做出决定，在这本新书里，不再介绍太多扩展库的应用，而是把重点放在 Python 语法和内置对象、标准库对象的高级应用上，注重 Python 的内涵，注重内功的修炼，尽量往纵深发展，争取用最简练的语言介绍那些作者认为比较高级的用法。在编写过程中，尽量减少与前几本书中内容的重复，补充大量新案例和高级用法。当然，前面几本书里的有些案例在这本书里又出现了，但是仔细的读者应该能发现，很多案例代码都进行了必要的改写和优化，更加 Pythonic，更加优雅和高效。自从答应了写这本书之后，作者在不影响正常教学和科研工作的情况下每天拿出至少 10 个小时查阅资料、编写代码或者整理书稿，前后用了一年左右的时间，现在回头想想也挺值得，在整理资料和编写案例代码的同时作者自己也进步了很多，对 Python 有了更加深入的认识和理解。

内容组织与阅读建议

全书共 13 章，面向 Python 3.5.x、Python 3.6.x 和 Python 3.7.x，重点关注 Python 内置对象和标准库对象的高级应用，以及比较前沿的、刚刚引入的一些新技术和新特性的用法，偶尔涉及一些扩展库用法。几乎每个知识点都配有大量的案例，把这些案例简单拼凑和集成就可以实现很多功能，实用性非常强。建议读者按章节顺序阅读，并且前后结合地反复阅读，不要随意跳过任何内容，或许不经意间会发现自己正需要的知识或者得到某种灵感。另外，虽然本书的定位是 Python 高级编程，但也同样适用于初学者，请初学者不要觉得有压力，如果有些地方暂时看不懂，可以先跳过去，或许过几天再看就明白了。当然，如果实在看不懂的话，及时和作者沟通应该会得到帮助。

第 1 章　管中窥豹：Python 概述。介绍 Python 语言的特点与主流版本，编程规范与代码优化建议，虚拟开发环境的创建与配置，扩展库的安装方法，开发与发布自己的包，Python 程序伪编译与打包，从命令行和外部文件获取配置信息。

第 2 章　万丈高楼平地起：运算符、表达式与内置对象。详细介绍 Python 的运算符与内置函数的用法，以及变量与常量的概念。

第 3 章　玄之又玄，众妙之门：详解 Python 序列结构。详解列表、元组、字典、集合等对象的特点与用法，列表推导式、生成器推导式、字典推导式与集合推导式，切片操作，序列解包，枚举、数组、队列、堆等常用结构用法。

第 4 章　反者，道之动：程序控制结构。详解 Python 中的选择结构和循环结构，else 的几种用法，选择结构的多种实现方式，break 与 continue 语句，循环结构的代码优化技巧。

第 5 章　代码复用技术(一)：函数。详解函数的定义与嵌套定义语法，可调用对象与

修饰器原理,位置参数、默认值参数、关键参数以及参数的序列解包,局部变量、全局变量与 nonlocal 变量,生成器函数,lambda 表达式,函数柯里化,泛型函数,协程函数,回调函数。

第 6 章　代码复用技术(二):面向对象程序设计。详解类的定义语法与实例化方法,数据成员、成员方法、属性以及静态方法与类方法,继承与多态,依赖注入技术的 Python 实现,特殊成员重写与运算符重载。

第 7 章　文本处理(一):字符串。详解字符串编码与字符串格式化方法、字符串对象方法、文本排版与压缩、分词、汉字拼音有关的技术。

第 8 章　文本处理(二):正则表达式。详解正则表达式基本语法与扩展语法,正则表达式模块 re 的用法,正则表达式对象与 match 对象的用法。

第 9 章　数据永久化:文件内容操作。详解内置函数 open()与上下文管理语句 with 的用法,文本文件与二进制文件的操作,Excel、Word、zip、rar 等常见二进制文件操作技术。

第 10 章　文件与文件夹操作。详解 os、os. path、shutil、glob、fnmatch 等模块在文件与文件夹操作方面的用法。

第 11 章　代码质量保障:异常处理结构、程序调试与测试。详解异常处理结构,文档测试与单元测试技术,覆盖测试与软件性能测试技术,IDLE、pdb、Eclipse+Pydev 等不同的代码调试技术。

第 12 章　多任务与并行处理:线程、进程、协程、分布式、GPU 加速。详解多线程与多进程编程技术,线程与进程的同步技术,不同进程间数据交换与共享技术,协程,spark 并行计算与 GPU 编程。

第 13 章　互通互联:asyncio 提供的网络通信功能。详解 asyncio 提供的网络通信功能,重点介绍 Transport、Protocol、StreamReader 以及 StreamWriter 等类的用法。

配套资源

本书提供所有案例源代码,可以登录清华大学出版社网站(www. tup. com. cn)下载,或加入本书读者群(QQ 群号 456324891,加入时请注明是读者,如果这个群满了的话会在群简介中给出下一个群号)下载最新配套资源并与作者交流,当然也欢迎关注微信公众号“Python 小屋”及时阅读作者写的最新案例代码,有些代码是在本书完稿之后新写的,是书上没有的,算作是一个很好的补充。

适用读者

- 已经具有一定 Python 水平的软件开发工程师。
- 打算深入探究 Python 高级编程的狂热爱好者。
- 各专业研究生、本科生、专科生的程序设计教材。
- 可能有些内容看起来会稍微有些吃力的其他 Python 初学者。

致谢

首先感谢父母的养育之恩，在当年那么艰苦的条件下还坚决支持我读书，没有让我像其他有的同龄的孩子一样辍学。感谢姐姐、姐夫多年来对我的爱护以及在老家对父母的照顾，感谢善良的弟弟、弟媳在老家对父母的照顾，正是有了你们，远离家乡的我才能安心工作。当然，最应该感谢的是妻子和孩子对我这个技术狂人的理解和宽容，这些年来她们已经习惯了正在吃饭的我突然萌发一个思路就放下饭碗去计算机前写代码的情景，习惯了我每个周末和假期都在教研室看书或写代码而不能陪她们玩，也习惯了周末的中午和晚上她们做好饭后再打电话催我回家。为了支持我的工作，她们还在我的几本书正式出版之前阅读了我的书稿。

感谢我的领导冯烟利教授提供了良好的教学科研环境，这样的工作环境让人觉得非常舒适，每个人都可以安心做好自己的事，发挥出最大潜力。

感谢选用 Python 程序设计系列教材的老师和长期关注微信公众号“Python 小屋”的朋友以及系列图书读者 QQ 群里的朋友，感谢你们指出前几本书里存在的几个小错误，和你们的交流也给了我很多启发！

感谢每一位读者，感谢您在茫茫书海中选择了本书，衷心希望您能够从本书中受益，学到真正需要的知识！同时也期待每一位读者的热心反馈，随时欢迎您指出书中的不足！

本书在编写出版过程中也得到清华大学出版社的大力支持和帮助，尤其是非常有远见的责任编辑白立军老师对这套 Python 系列图书的策划，一并表示衷心的感谢。

董付国　于山东烟台

2017 年 3 月

目　录



CONTENTS

第 1 章　管中窥豹：Python 概述 …… 1

1.1　Python 是这样一种语言 …… 1
1.2　Python 版本之争 …… 1
1.3　Python 编程规范与代码优化建议 …… 2
1.4　Python 虚拟开发环境的搭建 …… 4
1.5　Eclipse+PyDev 环境搭建和使用 …… 4
1.6　安装扩展库的几种方法 …… 6
1.7　标准库与扩展库中对象的导入与使用 …… 7
1.7.1　import 模块名 [as 别名] …… 8
1.7.2　from 模块名 import 对象名[as 别名] …… 8
1.7.3　from 模块名 import* …… 9
1.7.4　模块导入时的搜索路径 …… 10
1.8　编写与发布自己的包 …… 10
1.9　Python 程序伪编译与打包 …… 12
1.10　从命令行参数和配置文件获取信息 …… 14

第 2 章　万丈高楼平地起：运算符、表达式与内置对象 …… 16

2.1　Python 常用内置对象 …… 16
2.1.1　常量与变量 …… 17
2.1.2　数字 …… 18
2.1.3　字符串 …… 21
2.1.4　列表、元组、字典、集合 …… 22
2.2　Python 运算符与表达式 …… 23
2.2.1　算术运算符 …… 24
2.2.2　关系运算符 …… 25
2.2.3　成员测试运算符 in 与同一性测试运算符 is …… 26
2.2.4　位运算符与集合运算符 …… 27
2.2.5　逻辑运算符 …… 28
2.2.6　矩阵乘法运算符@ …… 29

2.2.7 补充说明 …… 29
2.3 Python 关键字简要说明 …… 30
2.4 Python 常用内置函数用法精要 …… 31
2.4.1 类型转换与类型判断 …… 34
2.4.2 最值与求和 …… 38
2.4.3 基本输入输出 …… 40
2.4.4 排序与逆序 …… 42
2.4.5 枚举与迭代 …… 43
2.4.6 map()、reduce()、filter() …… 45
2.4.7 range() …… 48
2.4.8 zip() …… 48
2.4.9 eval()、exec() …… 49

第 3 章 玄之又玄，众妙之门：详解 Python 序列结构 …… 51

3.1 列表：打了激素的数组 …… 51
3.1.1 列表创建与删除 …… 52
3.1.2 列表元素访问 …… 53
3.1.3 列表常用方法 …… 54
3.1.4 列表对象支持的运算符 …… 59
3.1.5 内置函数对列表的操作 …… 61
3.1.6 使用列表模拟向量运算 …… 62
3.1.7 列表推导式语法与应用案例 …… 63
3.1.8 切片操作的强大功能 …… 70
3.2 元组：轻量级列表 …… 73
3.2.1 元组创建与元素访问 …… 73
3.2.2 元组与列表的异同点 …… 74
3.2.3 生成器推导式 …… 75
3.3 字典：反映对应关系的映射类型 …… 76
3.3.1 字典创建与删除 …… 77
3.3.2 字典元素的访问 …… 77
3.3.3 元素的添加、修改与删除 …… 79
3.3.4 标准库 collections 中与字典有关的类 …… 80
3.4 集合：元素之间不允许重复 …… 81
3.4.1 集合对象的创建与删除 …… 81
3.4.2 集合操作与运算 …… 82
3.4.3 不可变集合 frozenset …… 84
3.4.4 集合应用案例 …… 84
3.5 序列解包的多种形式和用法 …… 86

3.6 标准库中的其他常用数据类型…… 88
3.6.1 枚举类型 …… 88
3.6.2 数组类型 …… 89
3.6.3 队列 …… 90
3.6.4 具名元组 …… 92
3.6.5 堆 …… 92

第 4 章 反者,道之动:程序控制结构 …… 94

4.1 条件表达式…… 94
4.2 选择结构…… 96
4.2.1 单分支选择结构 …… 96
4.2.2 双分支选择结构 …… 97
4.2.3 多分支选择结构 …… 98
4.2.4 选择结构的嵌套 …… 99
4.2.5 构建跳转表实现多分支选择结构…… 100
4.3 循环结构 …… 100
4.3.1 for 循环与 while 循环 …… 100
4.3.2 break 与 continue 语句 …… 101
4.3.3 循环代码优化技巧…… 102
4.4 精彩案例赏析 …… 103

第 5 章 代码复用技术(一):函数 …… 111

5.1 函数定义与使用 …… 111
5.1.1 基本语法…… 111
5.1.2 函数嵌套定义、可调用对象与修饰器 …… 113
5.1.3 函数对象成员的动态性…… 117
5.1.4 函数递归调用…… 117
5.2 函数参数 …… 118
5.2.1 位置参数…… 120
5.2.2 默认值参数…… 120
5.2.3 关键参数…… 122
5.2.4 可变长度参数…… 122
5.2.5 强制函数的某些参数必须以关键参数形式进行传值…… 123
5.2.6 强制函数的所有参数必须以位置参数形式进行传值…… 124
5.2.7 传递参数时的序列解包…… 125
5.2.8 标注函数参数与返回值类型…… 126
5.3 变量作用域 …… 127
5.3.1 全局变量与局部变量…… 127

5.3.2 nonlocal 变量 …… 130

5.4 lambda 表达式 …… 130

5.5 生成器函数设计要点 …… 133

5.6 偏函数与函数柯里化 …… 135

5.7 单分发器与泛型函数 …… 136

5.8 协程函数 …… 138

5.9 注册程序退出时必须执行的函数 …… 140

5.10 回调函数原理与实现 …… 140

5.11 精彩案例赏析 …… 141

第 6 章 代码复用技术(二)：面向对象程序设计 …… 167

6.1 类的定义与使用 …… 167

6.1.1 基本语法 …… 167

6.1.2 type 类 …… 168

6.1.3 定义带修饰器的类 …… 169

6.2 数据成员与成员方法 …… 171

6.2.1 私有成员与公有成员 …… 171

6.2.2 数据成员 …… 172

6.2.3 成员方法、类方法、静态方法、抽象方法 …… 173

6.2.4 属性 …… 175

6.2.5 类与对象的动态性、混入机制 …… 177

6.3 继承、多态、依赖注入 …… 179

6.3.1 继承 …… 179

6.3.2 多态 …… 181

6.3.3 依赖注入技术的不同实现方法 …… 182

6.4 特殊方法与运算符重载 …… 185

6.5 精彩案例赏析 …… 187

6.5.1 自定义队列 …… 187

6.5.2 自定义栈 …… 191

6.5.3 自定义集合 …… 193

6.5.4 自定义数组 …… 199

6.5.5 自定义双链表 …… 204

6.5.6 自定义常量类 …… 206

6.5.7 自定义不允许修改值的字典 …… 207

6.5.8 自定义支持 with 关键字的类 …… 208

第 7 章 文本处理(一)：字符串 …… 209

7.1 字符串编码格式简介 …… 210

7.2 转义字符与原始字符串 …… 211
7.3 字符串格式化 …… 212
7.3.1 使用%符号进行格式化 …… 212
7.3.2 使用 format()方法进行字符串格式化 …… 213
7.3.3 格式化的字符串常量 …… 215
7.3.4 使用 Template 模板进行格式化 …… 215
7.4 字符串常用操作 …… 216
7.4.1 find()、rfind()、index()、rindex()、count() …… 216
7.4.2 split()、rsplit()、partition()、rpartition() …… 217
7.4.3 join() …… 219
7.4.4 lower()、upper()、capitalize()、title()、swapcase() …… 220
7.4.5 replace()、maketrans()、translate() …… 220
7.4.6 strip()、rstrip()、lstrip() …… 221
7.4.7 startswith()、endswith() …… 222
7.4.8 isalnum()、isalpha()、isdigit()、isdecimal()、isnumeric()、isspace()、isupper()、islower() …… 222
7.4.9 center()、ljust()、rjust()、zfill() …… 223
7.4.10 字符串对象支持的运算符 …… 224
7.4.11 适用于字符串对象的内置函数 …… 226
7.4.12 字符串对象的切片操作 …… 228
7.5 其他有关模块 …… 228
7.5.1 textwrap 模块 …… 228
7.5.2 zlib 模块提供的压缩功能 …… 231
7.6 字符串常量 …… 232
7.7 可变字符串 …… 232
7.8 中英文分词 …… 234
7.9 汉字到拼音的转换 …… 235
7.10 精彩案例赏析 …… 236

第8章 文本处理(二)：正则表达式 …… 239

8.1 正则表达式语法 …… 239
8.1.1 正则表达式基本语法 …… 239
8.1.2 正则表达式扩展语法 …… 240
8.1.3 正则表达式锦集 …… 241
8.2 直接使用正则表达式模块 re 处理字符串 …… 242
8.3 使用正则表达式对象处理字符串 …… 246
8.4 match 对象 …… 248
8.5 精彩案例赏析 …… 250

第 9 章　数据永久化：文件内容操作 …… 257

9.1　文件操作基本知识 …… 258
9.1.1　内置函数 open() …… 258
9.1.2　文件对象属性与常用方法 …… 260
9.1.3　上下文管理语句 with …… 261
9.2　文本文件内容操作案例精选 …… 261
9.3　二进制文件操作案例精选 …… 266
9.3.1　使用 pickle 模块读写二进制文件 …… 267
9.3.2　使用 struct 模块读写二进制文件 …… 268
9.3.3　使用 shelve 模块操作二进制文件 …… 269
9.3.4　使用 marshal 模块操作二进制文件 …… 270
9.3.5　其他常见类型二进制文件操作案例 …… 271

第 10 章　文件与文件夹操作 …… 281

10.1　os 模块 …… 281
10.2　os.path 模块 …… 284
10.3　shutil 模块 …… 286
10.4　其他常用模块 …… 287
10.4.1　glob 模块 …… 287
10.4.2　fnmatch 模块 …… 288
10.5　精彩案例赏析 …… 289

第 11 章　代码质量保障：异常处理结构、程序调试与测试 …… 293

11.1　异常处理结构 …… 293
11.1.1　异常的概念与表现形式 …… 293
11.1.2　Python 内置异常类层次结构 …… 294
11.1.3　异常处理结构 …… 296
11.1.4　断言与上下文管理语句 …… 301
11.2　文档测试 doctest …… 301
11.3　单元测试 unittest …… 304
11.4　覆盖测试 …… 307
11.5　软件性能测试 …… 308
11.6　代码调试 …… 310
11.6.1　使用 IDLE 调试 …… 310
11.6.2　使用 Eclipse+PyDev 进行代码调试 …… 311
11.6.3　使用 pdb 调试 …… 313

第 12 章 多任务与并行处理：线程、进程、协程、分布式、GPU 加速 …… 318

12.1 多线程编程 …… 318
12.1.1 线程概念与标准库 threading …… 318
12.1.2 线程对象 …… 320
12.1.3 线程调度 …… 323
12.1.4 Lock/RLock 对象 …… 324
12.1.5 Condition 对象 …… 326
12.1.6 Queue 对象 …… 328
12.1.7 Event 对象 …… 332
12.1.8 Semaphore 与 BoundedSemaphore …… 333
12.1.9 Barrier 对象 …… 334
12.2 多进程编程 …… 335
12.2.1 进程创建与管理 …… 335
12.2.2 进程同步技术 …… 336
12.2.3 Pool 对象 …… 337
12.2.4 Manager 对象 …… 341
12.2.5 Listener 与 Client 对象 …… 345
12.2.6 进程间数据交换与共享 …… 345
12.2.7 标准库 subprocess …… 348
12.3 协程 …… 349
12.4 concurrent.futures 模块提供的并发执行功能 …… 351
12.5 pySpark 并行计算与分布式计算框架 …… 353
12.6 GPU 编程 …… 359
12.6.1 使用 pycuda 实现 GPU 加速 …… 359
12.6.2 使用 pyopencl 实现 GPU 加速 …… 360
12.6.3 使用 tensorflow 实现 GPU 加速 …… 362

第 13 章 互通互联：asyncio 提供的网络通信功能 …… 364

13.1 Transport 类与 Protocol 类 …… 364
13.2 StreamReader 与 StreamWriter …… 369

附录 精彩在继续 …… 373

附录 A GUI 开发 …… 373
附录 B SQLite 数据库操作 …… 376
附录 C 计算机图形学编程 …… 377
附录 D 图像编程 …… 380
附录 E 数据分析、机器学习、科学计算可视化 …… 383

附录 F　密码学编程 …… 383
附录 G　系统运维 …… 384
附录 H　Windows 系统编程 …… 384
附录 I　软件分析与逆向工程 …… 386

参考文献 …… 388

第 1 章



管中窥豹：Python 概述

1.1　Python 是这样一种语言

有不少人说 Python 是一种“大蟒蛇语言”。虽然在英语中 Python 确实有大蟒蛇的意思，但 Python 语言和大蟒蛇却没有任何关系。Python 语言的名字来自于一个著名的电视剧 *Monty Python's Flying Circus*，Python 之父 Guido van Rossum 是这部电视剧的狂热爱好者，所以把他设计的语言命名为 Python。

也有人说 Python 是一门脚本语言，这也是不准确的，远远不足以反映 Python 的强大。Python 并不仅仅是一门脚本语言，更是一门跨平台、开源、免费的解释型高级动态编程语言，是一种通用编程语言。除了可以解释执行之外，Python 还支持将源代码伪编译为字节码来优化程序提高运行速度并对源代码进行保密，也支持使用 py2exe、pyinstaller、cx_Freeze 或其他类似工具将 Python 程序及其所有依赖库打包成为各种平台上的可执行文件，当然也包括扩展名为 exe 的 Windows 可执行程序，从而可以脱离 Python 解释器环境和相关依赖库，能够在 Windows 平台上独立运行，并且还支持制作成.msi 安装包；Python 支持命令式编程（How to do）和函数式编程（What to do）两种方式，完全支持面向对象程序设计（虽然并不强制要求处处体现面向对象编程的思想和有关特征，但实际上人们无时无刻不在使用），语法简洁清晰，功能强大且易学易用，更重要的是拥有大量的几乎支持所有领域应用开发的成熟扩展库和狂热支持者。

当然，也有人喜欢把 Python 称为“胶水语言”，这确实是 Python 的重要特点之一，它可以把多种不同语言编写的程序融合到一起实现无缝拼接，更好地发挥不同语言和工具的优势，满足不同应用领域的需求。

1.2　Python 版本之争

众所周知，Python 官方网站同时发行和维护着 Python 2.x 和 Python 3.x 两个不同系列的版本，并且版本更新速度非常快（6 个月左右更新一次小版本号）。目前最新版本分别是 Python 2.7.13、Python 3.4.6、Python 3.5.3 和 Python 3.6.1，Python 3.7 已在研发中，估计很快就会推出。Python 2.x 和 Python 3.x 这两个系列的版本之间很多用法是不兼容的（让人欣慰的是，除了一些新特性、运算符和标准库对象之外，同一个系列的不同版本之间绝大多数用法是完全一致的），除了基本输入输出方式有所不同，很多内置函

数和标准库对象的用法也有非常大的区别，Python 3. x 在增加了很多新标准库的同时也删除了一些 Python 2. x 的标准库，还有些 Python 2. x 的标准库在 Python 3. x 中进行了合并和拆分。当然，适用于 Python 2. x 和 Python 3. x 的扩展库之间更是差别巨大，这应该是现有系统进行版本迁移时最大的障碍。因此，在正式开始使用 Python 之前，必须要选择合适的版本，以免浪费时间。

在选择 Python 版本的时候，一定要先考虑清楚自己学习 Python 的目的是什么，打算做哪方面的开发，该领域或方向有哪些扩展库可用，这些扩展库最高支持哪个版本的 Python，是否还在维护和更新。这些问题全部明确以后，再最终确定选择哪个版本，这样才能事半功倍，而不至于把太多时间浪费在 Python 以及各种扩展库的反复安装和卸载上，虽然这并不是非常麻烦。另外，当较新的 Python 版本推出之后，不要急于安装，而是应该在确定自己所必须使用的扩展库也推出了与之匹配的稳定版本之后再一起进行更新。

总体来看，Python 3. x 的设计理念更加合理、高效和人性化，全面普及和应用是必然的，越来越多的扩展库也以非常快的速度推出了与最新 Python 版本相适应的版本。如果暂时还没想到要做什么行业领域的应用开发，或者仅仅是为了尝试一种新的、好玩的语言，那么请毫不犹豫地选择 Python 3. x 系列的最高版本。

1.3　Python 编程规范与代码优化建议

没有规矩，不成方圆。任何一种语言都有一些约定俗成的编码规范，Python 也不例外。Python 非常重视代码的可读性，对代码布局和排版有更加严格的要求。虽然一些大型软件公司对自己公司程序员编写的代码在布局、结构、标识符命名等方面有一些特殊的要求，但其中很多思想是相同的，目的也是一致的。这里重点介绍 Python 社区对代码编写的一些共同的要求、规范和一些常用的代码优化建议，最好在开始编写第一段代码的时候就要遵循这些规范和建议，养成一个好的习惯。

（1）严格使用缩进来体现代码的逻辑从属关系。Python 对代码缩进是硬性要求，这一点必须时刻注意。如果某个代码段的缩进不对，那么整个程序就是错的，要么是语法错误无法执行，要么是逻辑错误导致错误结果，而检查这样的错误会花费很多时间。

（2）每个 import 语句只导入一个模块，最好按标准库、扩展库、自定义库的顺序依次导入。尽量避免导入整个库，最好只导入确实需要使用的对象，这会让程序运行更快。

（3）最好在每个类、函数定义和一段完整的功能代码之后增加一个空行，在运算符两侧各增加一个空格，逗号后面增加一个空格。按照这样的规范写出来的代码布局和排版比较松散，阅读起来更加轻松。不论是前面第一条讲的缩进，还是这里谈的空行与空格，主要是提高代码可读性，正如 The Zen of Python 所说：Sparse is better than dense、Readability counts。稍微有点例外的是，在正常的赋值表达式中等号两侧都是各增加一个空格，但在定义函数的默认值参数和使用关键参数调用函数时一般并不在参数赋值的等号两侧增加空格。这样松中有紧也是为了提高代码的可读性，正所谓“张而不弛，文武弗能也；弛而不张，文武弗为也；一张一弛，文武之道也。”



（4）尽量不要写过长的语句。如果语句过长，可以考虑拆分成多个短一些的语句，以保证代码具有较好的可读性。如果语句确实太长而超过屏幕宽度，最好使用续行符（line continuation character）“\”，或者使用圆括号将多行代码括起来表示是一条语句。

（5）虽然 Python 运算符有明确的优先级，但对于复杂的表达式建议在适当的位置使用括号使得各种运算的隶属关系和顺序更加明确，正如 The Zen of Python 所说：Explicit is better than implicit。

（6）对关键代码和重要的业务逻辑代码进行必要的注释。统计数据表明，一个可读性较好的程序中应包含大概 30% 以上的注释。在 Python 中有两种常用的注释形式：#和三引号。#用于单行注释，三引号常用于大段说明性文本的注释。

（7）在开发速度和运行速度之间尽量取得最佳平衡。内置对象运行速度最快，标准库对象次之，用 C 或 FORTRAN 编写的扩展库速度也比较快，而纯 Python 的扩展库往往速度慢一些。因此，在开发项目时，应优先使用 Python 内置对象，其次考虑使用 Python 标准库提供的对象，最后考虑使用第三方扩展库。然而，有时候只使用内置对象和标准库对象的话，很可能无法直接满足需要。这时候有两个选择：一是使用内置对象和标准库对象编写代码实现特定的逻辑；二是使用合适的扩展库对象。至于如何取舍，最终还是取决于业务逻辑的复杂程度和对运行速度的要求这两者之间的平衡。

（8）根据运算特点选择最合适的数据类型来提高程序的运行效率。如果定义一些数据只是用来频繁遍历，最好优先考虑元组或集合。如果需要频繁地测试一个元素是否存在于一个序列中并且不关心其位置，尽量采用字典或者集合。列表和元组的 in 操作的时间复杂度是线性的，而对于集合和字典却是常数级的，与问题规模几乎无关。在所有内置数据类型中，列表的功能最强大，但开销也最大，运行速度最慢，应慎重使用。作为建议，应优先考虑使用集合和字典，元组次之，最后再考虑列表和字符串。

（9）充分利用关系运算符以及逻辑运算符 and 和 or 的惰性求值特点，合理组织条件表达式中多个条件的先后顺序，减少不必要的计算。

（10）充分利用生成器对象或类似迭代对象的惰性计算特点，尽量避免将其转换为列表、元组等类型，这样可以减少对内存的占用，降低空间复杂度。

（11）减少内循环中的无关计算，尽量往外层提取。

有很多成熟的工具可以检查 Python 代码的规范性，例如 pep8、flake8、pylint 等。可以使用 pip 来安装 pep8 工具，然后使用命令 pep8 test. py 来检查 test. py 文件中 Python 代码的规范性。pep8 常用的可选参数有--show-source、--first、--show-pep8 等。flake8 结合了 pyflakes 和 pep8 的特点，可以检查更多的内容，优先推荐使用，使用 pip install flake8 可以直接安装，然后使用命令 flake8 test. py 检查 test. py 中代码的规范性。也可以使用 pip 安装 pylint，然后使用命令行工具 pylint 或者可视化工具 pylint-gui 来检查程序的规范性。

1.4　Python 虚拟开发环境的搭建

Python 支持创建多个虚拟环境，每个虚拟环境都是包含 Python 主程序和相应扩展库

的一个文件夹，每个虚拟环境都是独立的，多个虚拟环境之间互相不干扰，可以安装不同版本的 Python 解释器和扩展库。下面通过一个实际的例子来演示如何创建和使用 Python 虚拟环境，首先进入命令提示符环境并切换至 Python 安装目录的 tools\Scripts 文件夹，然后执行下面的命令：

```
..\..\python pyvenv.py ..\..\Python_docx
```

稍等片刻，当再次出现命令提示符的时候就表明 Python 虚拟环境创建成功了，接下来使用 cd 命令切换至 Python_docx\Scripts 文件夹中，执行 activate 命令，如成功就会发现前面的提示符有些变化。Python 虚拟环境的创建和使用主要步骤如图 1-1 所示。

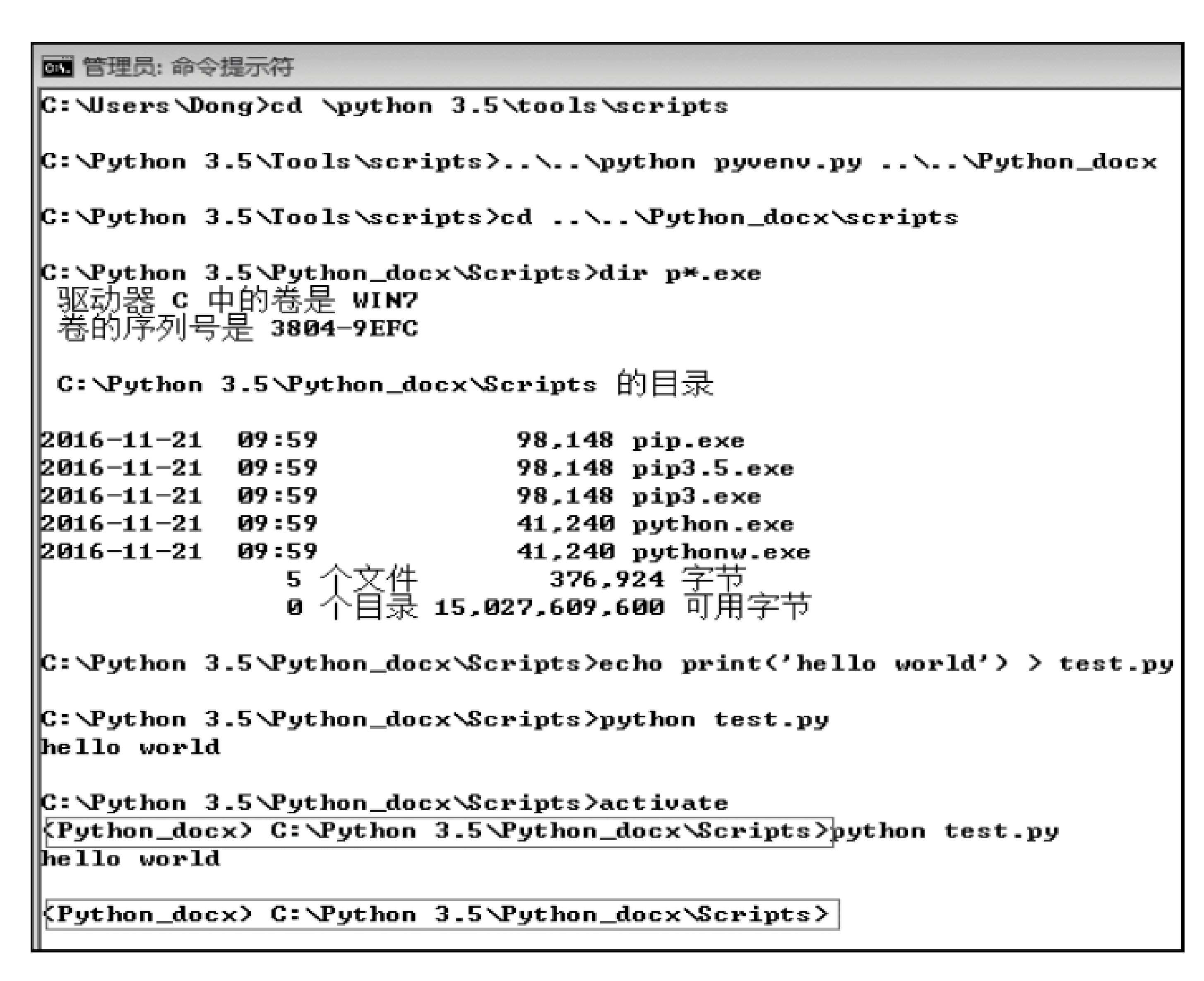


图 1-1　Python 虚拟环境创建和使用

1.5　Eclipse + PyDev 环境搭建和使用

其实作者更喜欢使用 IDLE 来写 Python 代码，之前的几本书上所有案例代码都是用 IDLE 写的，作者开发的一套“边讲边练类课程教学管理系统”也是完全用 IDLE 写的。其实，Python 解释器 python. exe 才是根本，PyCharm、wingIDE、PythonWin、Eclipse + PyDev、Eric 或其他 IDE 开发环境不过是对其进行了一定的封装，还有些像 Anaconda、Python(x,y)、zwPython 之类的开发环境集成了一些常用的或者特定领域的扩展库，所有这些都是为了使得安装和开发过程更加愉快而已，当然这一点也很重要。



为了让自己表现得像一个 Python 高手，本书有些代码作者选择使用 Eclipse + PyDev，虽然这在很多时候并没有必要，也显示不出这个开发环境的优势，但不得不说，在开发大型项目时，这些带着外挂的开发环境确实比 IDLE 方便一些，尤其是标识符的自动识别和提醒功能可以减少很多拼写错误，而这在实际开发中是经常出现的一种错误。书中一些演示性和验证性的小代码片段仍使用了 IDLE，当然大家也可以使用自己喜欢的开发环境，作者觉得没必要在这个问题上纠结。

我们有理由相信，能选择和阅读本书的朋友应该已经具有了一定的计算机应用水平，完全可以非常顺利地安装和配置 Eclipse + PyDev 环境，所以这里只把关键的步骤介绍一下。

首先安装合适版本的 Eclipse 和 Java JDK，然后打开 Eclipse，依次单击菜单 Help→Install New Software，在弹出的窗口中单击 Add 按钮，在弹出的 Add Repository 窗口中的 Name 中填写 PyDev，在 Location 中填写 http://pydev.org/updates，如图 1-2 所示。接下来单击 OK 按钮后再单击 Next 按钮进行安装即可。

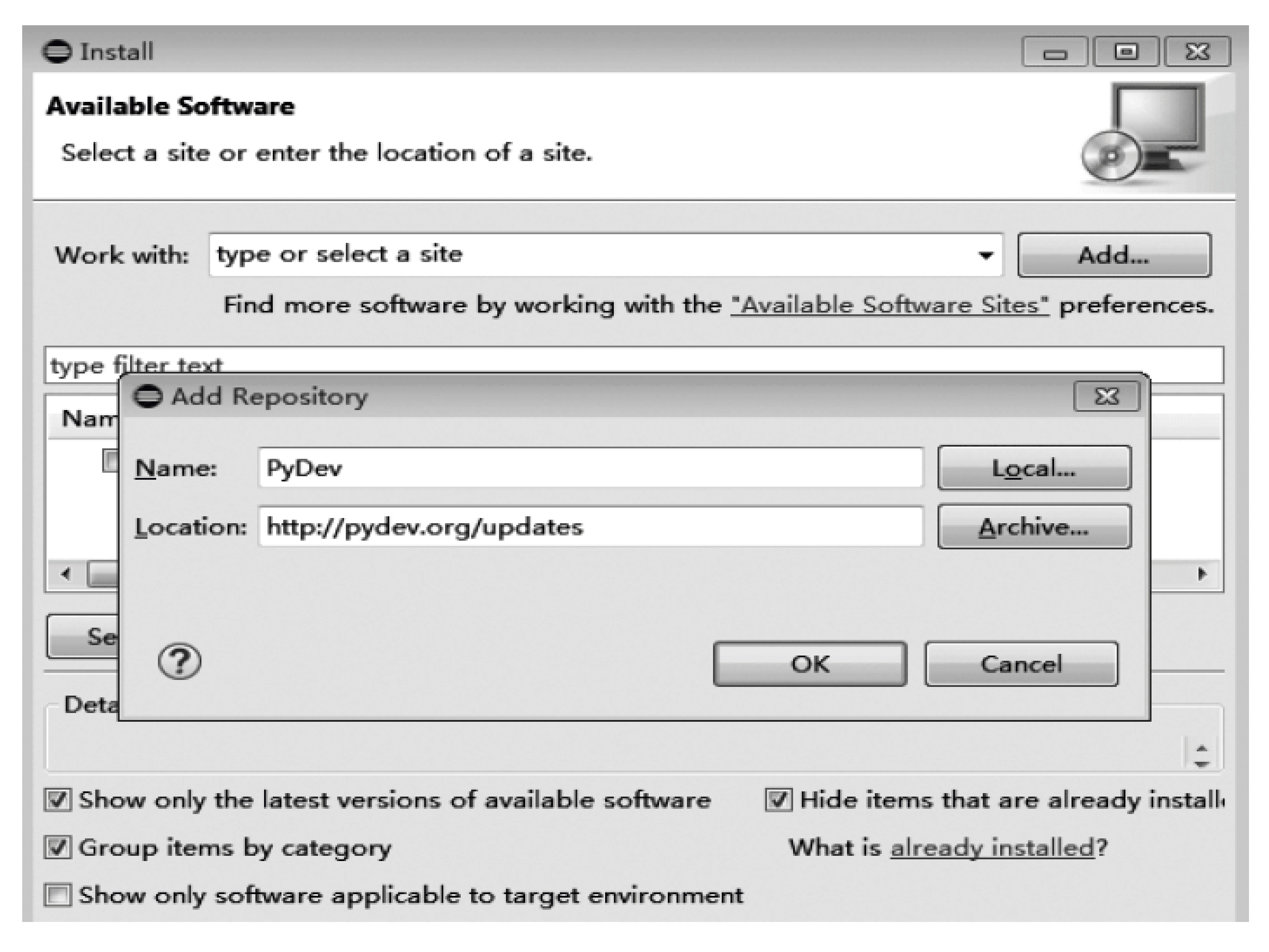


图 1-2　安装 PyDev

安装完之后需要进行简单的配置才能更好地使用这个环境，依次单击菜单 Window→Preferences 打开配置窗口，然后找到 PyDev 展开后进行必要的配置，图 1-3 中展示了如何配置 Python 解释器版本，这是非常重要的一项配置，其他如代码颜色、风格以及编辑器的有关配置请读者自行查阅资料，然后根据自己的喜好进行配置。

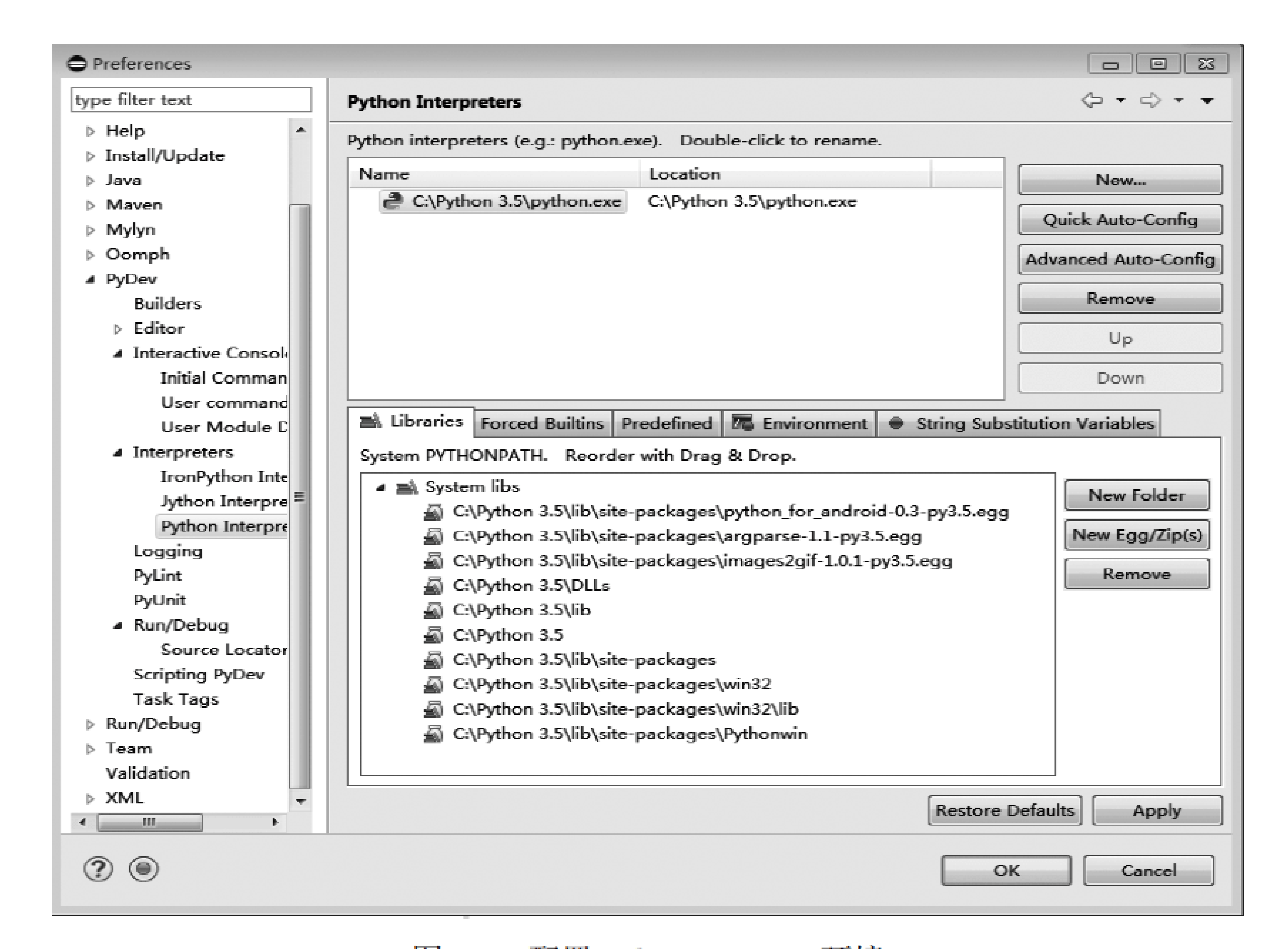


图 1-3　配置 Eclipse + PyDev 环境

1.6　安装扩展库的几种方法

除了使用源码安装和二进制安装包(并不是所有扩展库都提供这种方式)以外,easy_install 和 pip 工具已经成为管理 Python 扩展库的主要方式,其中 pip 用得更多一些。使用 pip 不仅可以查看本机已安装的 Python 扩展库列表,还支持 Python 扩展库的安装、升级和卸载等操作。使用 pip 工具管理 Python 扩展库只需要在保证计算机联网的情况下输入几个命令即可完成,极大方便了用户。常用 pip 命令的使用方法如表 1-1 所示。

表 1-1　常用 pip 命令使用方法

pip 命令示例	说　　明
pip download SomePackage[= = version]	下载扩展库的指定版本,不安装
pip freeze	以 requirements 的格式列出已安装模块
pip list	列出当前已安装的所有模块



续表

pip 命令示例	说　　明
pip install SomePackage[= = version]	在线安装 SomePackage 模块的指定版本
pip install SomePackage. whl	通过 whl 文件离线安装扩展库
pip install package1 package2…	依次(在线)安装 package1、package2 等扩展模块
pip install -r requirements. txt	安装 requirements. txt 文件中指定的扩展库
pip install --upgrade SomePackage	升级 SomePackage 模块
pip uninstall SomePackage[= = version]	卸载 SomePackage 模块的指定版本

在 https://pypi. python. org/pypi 中可以获得一个 Python 扩展库的综合列表，可以根据需要下载源码进行安装或者使用 pip 工具进行在线安装，也有一些扩展库还提供了. whl 文件和. exe 文件，大幅度简化了扩展库的安装过程。有些扩展库安装时要求本机已安装相应版本的 C/C ++ 编译器，或者有些扩展库暂时还没有与本机 Python 版本对应的官方版本，这时可以从 http://www. lfd. uci. edu/ ~ gohlke/pythonlibs/下载对应的. whl 文件(注意，不要修改文件名)，然后在命令提示符环境中使用 pip 命令进行安装。例如：

```
pip install pygame-1.9.2a0-cp35-none-win_amd64.whl
```

一般来讲，使用 pip 工具在线安装总是会自动选择扩展库的最新版本，但有时候会出现新版本与其他扩展库不兼容的情况，或者其他扩展库依赖待安装扩展库的较低版本，这时可以明确指定扩展库的版本号，例如：

```
pip install requests==2.12.4
```

如果需要安装的扩展库比较多，并且对版本号要求严格，可以使用类似于 pip install -r requirements. txt 这样的命令从 requirements. txt 文件中读取所需安装的扩展库信息并自动安装。这个 requirements. txt 可以手工编辑，也可以使用 pip freeze > requirements. txt 命令把本机已安装模块的信息快速生成为 requirements. txt 文件。在命令提示符环境中直接执行 pip 命令可以查看其他子命令，例如 wheel。然后执行 pip wheel -h 可以查看子命令的详细用法，不再赘述。

1.7　标准库与扩展库中对象的导入与使用

Python 默认安装仅包含基本或核心模块，启动时也仅加载了基本模块，在需要时再显式地导入和加载标准库和第三方扩展库，这样可以减小程序运行的压力，并且具有很强的可扩展性。从“木桶原理”的角度来看，这样的设计与安全配置时遵循的“最小权限”原则是一致的，也有助于提高系统安全性。如果需要的话，可以使用 sys. modules. items( ) 显示所有预加载模块的相关信息。另外，可以使用 sys. builtin_module_names 查看那些编译进 Python 解释器的模块名字，这些模块具有最快的访问速度和最高的执行效率。

当以模块形式导入. py 或. pyw 源文件时，系统会自动检查__pycache__文件夹中是否

存在相应的.pyc 文件,如果有且签名一致则自动从.pyc 文件导入以提高加载速度,如果源文件具有比.pyc 文件更新的签名,那么导入时会自动重新生成.pyc 文件。但需注意的是,如果源文件不存在,导入时不会检查__pycache__文件夹。因此,如果想实现无源码部署,应该把所有.pyc 文件放到源文件的文件夹中而不是__pycache__文件夹中。

1.7.1 import 模块名[as 别名]

使用这种方式导入以后,使用时需要在对象之前加上模块名作为前缀,必须以“模块名.对象名”的形式进行访问。如果模块名字很长的话,可以为导入的模块设置一个别名,然后使用“别名.对象名”的方式来使用其中的对象。

```
>>>import math                             #导入标准库 math
>>>math.sin(0.5)                           #求 0.5(单位是弧度)的正弦
0.479425538604203
>>>import random                           #导入标准库 random
>>>n =random.random()                      #获得[0,1) 内的随机小数
>>>n =random.randint(1,100)                #获得[1,100]区间上的随机整数
>>>n =random.randrange(1,100)              #返回[1,100)区间中的随机整数
>>>import os.path as path                  #导入标准库 os.path,并设置别名为 path
>>>path.isfile(r'C:\windows\notepad.exe')
True
>>>import numpy as np                      #导入扩展库 numpy,并设置别名为 np
>>>a =np.array((1,2,3,4))                  #通过模块的别名来访问其中的对象
>>>a
array([1,2,3,4])
>>>print(a)
[1 2 3 4]
```

1.7.2 from 模块名 import 对象名[as 别名]

使用这种方式仅导入明确指定的对象,并且可以为导入的对象确定一个别名。这种导入方式可以减少查询次数,提高访问速度,同时也可以减少程序员需要输入的代码量,不需要使用模块名作为前缀。

```
>>>from math import sin                    #只导入模块中的指定对象
>>>sin(3)
0.1411200080598672
>>>from math import sin as f               #给导入的对象起个别名
>>>f(3)
0.1411200080598672
>>>from os.path import isfile
>>>isfile(r'C:\windows\notepad.exe')
True
```



1.7.3　from 模块名 import *

这是上面用法的一种极端情况，可以一次导入模块中通过__all__变量指定的所有对象。

```
>>>from math import *                    #导入标准库 math 中所有对象
>>>gcd(36,18)                            #最大公约数
18
>>>pi                                    #常数 π
3.141592653589793
>>>e                                     #常数 e
2.718281828459045
>>>log2(8)                               #计算以 2 为底的对数值
3.0
>>>log10(100)                            #计算以 10 为底的对数值
2.0
>>>radians(180)                          #把角度转换为弧度
3.141592653589793
```

这种方式简单粗暴，写起来也比较省事，可以直接使用模块中的所有对象而不需要再使用模块名作为前缀。但一般并不推荐这样使用。一方面这样会降低代码的可读性，有时候很难区分自定义函数和从模块中导入的函数；另一方面，这种导入对象的方式将会导致命名空间的混乱。如果多个模块中有同名的对象，只有最后一个导入的模块中的对象是有效的，而之前导入的模块中的同名对象都将无法访问，不利于代码的理解和维护。例如，a. py 文件中内容如下：

```
def test():
    print('test in a.py')
```

b. py 文件中内容如下：

```
def test():
    print('test in b.py')
```

导入 a 模块以后，test( ) 方法是可用的，而导入 b 模块之后 a 模块中的 test( ) 方法就无法使用了。例如：

```
>>>from a import *
>>>test()
test in a.py
>>>from b import *                       #这会导致 a 模块中的 test()函数无法使用
>>>test()
test in b.py
```

1.7.4 模块导入时的搜索路径

不管以哪种形式导入模块并使用其中的对象,Python 首先在当前目录中查找模块文件,如果没有找到则从 sys 模块的 path 变量所指定的目录中查找,如果仍没有找到则抛出异常提示模块不存在。可以查看 sys 模块中 path 变量的值来获知 Python 导入模块时搜索模块的路径,也可以使用 append()方法向其中添加自定义的文件夹以扩展搜索路径。另外,在导入模块时,会优先导入相应的.pyc 文件(.py 或.pyw 为编译后生成的字节码文件),如果相应的.pyc 文件与.py 文件时间不相符或不存在对应的.pyc 文件,则导入.py 文件,同时生成.pyc 文件。

Python 还支持从 zip 文件中导入模块。假设当前文件夹中有个内含 Vector3.py 文件的 testZip.zip 文件,首先导入 sys 模块,然后执行 sys.path.append('testZip.zip'),然后即可导入 Vector3.py 文件作为模块来使用。

```
>>>import sys
>>>sys.path.append('testZip.zip')
>>>import Vector3
>>>Vector3.__file__                    #查看已导入模块对应的程序文件
'testZip.zip\\Vector3.py'
```

最后,按照 Python 编码规范,一般建议每个 import 语句只导入一个模块,并且要按照标准库、扩展库、自定义库的顺序进行导入。

1.8 编写与发布自己的包

除了可以在开发环境中或命令提示符环境中直接运行,任何 Python 程序文件都可以作为模块导入并使用其中的对象,这也是实现代码复用的重要形式。通过 Python 程序的__name__属性可以识别程序的使用方式,每个 Python 脚本在运行时都会有一个__name__属性,如果脚本作为模块被导入,则其__name__属性的值被自动设置为模块名;如果脚本作为程序直接运行,则其__name__属性值被自动设置为字符串"__main__"。例如,假设程序 hello.py 中的代码如下:

```
def main():                            #def 是用来定义函数的 Python 关键字
    if __name__=='__main__':           #选择结构,识别当前的运行方式
        print('This program is run directly.')
    elif __name__=='hello':            #冒号、换行、缩进表示一个语句块的开始
        print('This program is used as a module.')

main()                                 #调用上面定义的函数
```

那么通过任何方式直接运行该程序时都会得到下面的结果:

```
This program is run directly.
```

而在使用 import hello 导入该模块时,得到结果如下:

```
This program is used as a module.
```

很明显,对于大型软件的开发,不可能把所有代码都存放到一个文件中,那样会使得代码很难维护。对于大型软件系统,可以使用包来管理多个模块。包是 Python 用来组织命名空间的重要方式,可以看作是包含大量 Python 程序模块的文件夹。在包的每个子文件夹中都必须包含一个__init__. py 文件,该文件可以是一个空文件,仅用于表示当前文件夹是一个包。__init__. py 文件的主要用途是设置__all__变量以及执行初始化包所需的代码,其中__all__变量中定义的对象可以在使用"from…import ＊"时全部被正确导入。

假设有如下结构的包:

```
sound/                        Top-level package
        __init__.py           Initialize the sound package
       formats/               Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
               ⋮
        effects/              Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
               ⋮
        filters/              Subpackage for filters
              __init__.py
              equalizer.py
                ⋮
```

那么,可以在自己的程序中使用下面的代码导入其中一个模块:

```
import sound.effects.echo
```

然后使用完整路径来访问其中的对象,例如:

```
sound.effects.echo.echofilter(input,output,delay=0.7,atten=4)
```

如果 sound\effects__init__. py 文件中有下面一行的代码:

```
__all__=['echo','surround','reverse']
```

那么就可以使用下面的方式来导入__all__变量指定的 echo、surround 和 reverse 对象:

```
from sound.effects import *
```

然后使用下面的方式来使用其中的成员:

```
echo.echofilter(input,output,delay=0.7,atten=4)
```

在组织自己的包时，应避免只有一个__init__.py 的文件夹。例如，如果可以用 common.py 实现同样的功能，就不要使用 common__init__.py 这样的结构，除非 common.py 需要实现的功能非常多，需要分散到多个库中，例如 common\database.py、common\net.py、common\user.py 等，然后再在 common__init__.py 中通过__all__变量指定哪些对象可以导入。

在自己编写模块时，可能需要反复修改代码然后重新导入模块进行测试，此时可以使用 imp 模块或 importlib 模块的 reload( )函数。重新加载模块时要求该模块已经被正确加载过，也就是说，第一次导入和加载模块时不能使用 reload( )方法。

本节的最后介绍如何把自己的 Python 包发布到 pypi，假设有 Python 程序文件 fastcopytree.py，现在编写一个对应的 setup.py 文件，内容如下：

```
from distutils.core import setup

setup(name='fastcopytree',version='1.0.0',py_modules=['fastcopytree'],
    author='dong fuguo',author_email='dongfuguo2005@126.com',
    url='http://user.qzone.qq.com/306467355/2')
```

然后在命令提示符环境中执行 python setup.py register 并选择 1 进行登录，如果没有 pypi 账号可以选 2 先进行注册，登录后再使用 python setup.py register -n 检查包名是否可用，最后使用 python setup.py sdist upload 命令把自己的包发布到 pypi 上去，别人就可以使用 pip install fastcopytree 来安装这个包了。当然，上面的 setup.py 程序还有更多的参数可以使用，例如，使用 python setup.py bdist_wininst 可以创建 Windows 安装包。

1.9 Python 程序伪编译与打包

众所周知，Python 是纯粹的自由软件，源代码和解释器 CPython 遵循 GPL(GNU General Public License)协议。那么很自然会有人有这样的疑问：难道 Python 程序只能以源代码的方式来运行吗？能不能通过某种方式来保护自己的源代码呢？答案是肯定的。这方面的技术主要有两种：一种方法是把 Python 程序伪编译成扩展名为 pyc 的字节码文件；一种是通过 py2exe、pyinstaller 或者 cx_Freeze 对 Python 程序进行打包成为不同平台上的二进制可执行文件。

1. Python 程序伪编译

可以使用 py_compile 模块的 compile( )函数或 compileall 模块的 compile_file( )函数对 Python 源程序文件进行伪编译得到扩展名为 pyc 的字节码以提高加载和运行速度，同时还可以隐藏源代码。假设有 Python 程序 Stack.py 文件，并已导入 py_compile 和 compileall 模块，那么常用的编译方法如下。

1) py_compile.compile('Stack.py')

等价于 compileall.compile_file('Stack.py')，也等价于在命令提示符环境中执行命令 python -m py_compile Stack.py，都会在__pycache__文件夹中生成文件 Stack.cpython-



35. pyc。

2）py_compile. compile('Stack. py', optimize = 1)

等价于 compileall. compile_file('Stack. py', optimize = 1)，也等价于在命令提示符环境中执行命令 python -O -m py_compile Stack. py，属于优化编译，会在__pycache__文件夹中生成 Stack. cpython-35. opt-1. pyc 文件。

3）py_compile. compile('Stack. py', optimize = 2)

等价于 compileall. compile_file('Stack. py', optimize = 2)，也等价于在命令提示符环境中执行命令 python -OO -m py_compile Stack. py，属于高级优化编译，会在__pycache__文件夹中生成文件 Stack. cpython-35. opt-2. pyc。

此外，Python 的 compileall 模块还提供了 compile_dir() 和 compile_path() 等方法，用来支持 Python 源程序文件的批量编译。

那么问题来了，是不是编译成. pyc 文件以后真的无法查看源代码呢？实际上，还是有很多办法可以查看的。例如，可以使用 Python 扩展库 uncompyle6 或其他类似模块来完成这个功能。使用 pip 工具安装 uncompyle6 之后，可以使用下面的代码对上面生成的. pyc 文件进行反编译得到源代码：

```
uncompyle6.uncompyle_file('__pycache__\\Stack.cpython-35.opt-1.pyc',
                          open('__pycache__\\Stack.py','w'))
```

另外，http://tool. lu/pyc/这个网站可以在线上传一个. pyc 文件，然后可以得到 Python 源代码（个别地方可能不是非常准确，需要自己稍做调整），并且还提供了一定的代码美化功能，能够自动处理代码布局和排版规范。

2. Python 程序打包

保护源代码的更好方式是把 Python 程序转换为二进制可执行程序之后再发布，这样还使得打包后的程序可以在没有安装 Python 环境和相应扩展库的系统中运行，极大地方便了用户。可以把 Python 程序打包为可执行程序的工具有 py2exe（仅适用于 Windows 平台）、pyinstaller、cx_Freeze 等。当然，打包之前首先应该保证 Python 程序可以正常运行，并且本机已安装了所有需要的扩展模块和相关的动态链接库文件。

以使用 py2exe 在 Windows 平台上打包 Python 程序为例，假设有 Python 源程序文件 test. py，然后编写 setup. py 文件，内容为

```
import distutils
import py2exe
distutils.core.setup(console=['test.py'])
```

然后在命令提示符下执行下面的命令：

```
python setup.py py2exe
```

接下来就会看到控制台窗口中大量的提示内容飞快地闪过，这个过程会自动搜集 test. py 程序执行所依赖的所有支持文件，如果创建成功的话则会在当前文件夹下生成一

个 dist 子文件夹,其中包含了最终程序执行所需要的所有内容。等待编译完成以后,将 dist 文件中的文件打包发布即可。当然,对于简单的 Python 程序,直接在命令提示符环境执行 build_exe test. py 就可以生成. exe 可执行文件了。

对于控制台应用程序,要想转换为 exe 可执行程序直接套用上面的代码框架即可,只需要把

```
distutils.core.setup(console=['test.py'])
```

这行代码中的文件名替换为自己的 Python 程序文件名即可。对于 GUI 应用程序,还需要将上面代码中的关键字 console 修改为 windows。

另外一个比较好用的 Python 程序打包工具是 pyinstaller,同时适用于 Windows 平台和 Linux 平台上 Python 程序的打包,使用 pip 工具安装之后在命令提示符环境中使用命令 pyinstaller -F -w kousuan. pyw 或者 python pyinstaller-script. py -F -w kousuan. pyw 即可将 Python 程序 kousuan. pyw 及其所有依赖包打包成为当前所用平台上的可执行文件。

如果使用 cx_Freeze 工具的话,假设有 Python 程序 hello. py,在命令提示符环境执行 python cxfreeze hello. py 即可快速创建可执行程序并自动搜集依赖的包。除此之外,在 Windows 平台上 cx_Freeze 还支持制作. msi 安装程序。

1.10 从命令行参数和配置文件获取信息

实际开发中,很多时候需要从外部获取数据,根据用户的输入或配置信息来决定下一步应采取的行为。除了使用内置函数 input( )或者 GUI 库的控件和对话框来接收用户输入以外,还可以使用 sys 和 argparse 模块来接收命令行参数,使用 configparser 模块从外部配置文件中获取信息。

sys 模块的 argv 是一个包含若干字符串的列表,用来接收命令行参数,其中第一个元素 argv[0]是程序本身的名字,后面其他元素是用户输入的其他参数。在输入时,多个命令行参数之间使用空格分隔。使用 argparse 模块接收并解析命令行参数的内容请参考第 12 章使用多线程批量复制文件的示例代码。

如果某个程序需要配置大量信息,那么可以把与程序有关的这些信息单独存放到一个配置文件中,这样就不用重复输入了,并且可以在不修改代码的前提下改变代码的行为,只需要提供不同的配置文件即可。configparser 模块提供了非常方便的配置文件读取接口,假设配置文件 test. ini 的内容如下:

```
[DEFAULT]
default1 =0
default2 =1
default3 =% (name)s,% (age)s,% (sex)s
name =default
age =18
sex =M
```




```
[SECTION1]
name=dong
age=39
sex=M
addr=yantai

[SECTION2]
name=zhang
age=40
sex=F
weight=50
```

那么可以通过下面的代码来读取和显示配置文件中的信息：

```
import configparser

conf=configparser.ConfigParser()
conf.read('test.ini')
print(conf.get('SECTION1','age'))
print(conf.get('SECTION2','sex'))
#使用 SECTION1 节中的信息替换 DEFAULT 节中的 default3 变量
print(conf.get('SECTION1','default3'))
print(conf.get('SECTION2','default3'))
print(conf.get('DEFAULT','default3'))
```

第 2 章



万丈高楼平地起：运算符、表达式与内置对象

2.1 Python 常用内置对象

对象是 Python 中最基本的概念之一，在 Python 中一切都是对象，除了整数、实数、复数、字符串、列表、元组、字典、集合，还有 zip、map、enumerate、filter 等对象，函数和类也是对象。常用的 Python 内置对象如表 2-1 所示。

表 2-1 Python 内置对象

对象类型	类型名称	示 例	简要说明
数字	int，float，complex	1234，3.14，1.3e5，3+4j	数字大小没有限制，内置支持复数及其运算
字符串	str	'swfu'，"I'm student"，'''Python'''，r'abc'，R'bcd'	使用单引号、双引号、三引号作为定界符，以字母 r 或 R 引导的表示原始字符串
字节串	bytes	b'hello world'	以字母 b 引导，可以使用单引号、双引号、三引号作为定界符
列表	list	[1，2，3]，['a'，'b'，['c'，2]]	所有元素放在一对方括号中，元素之间使用逗号分隔，其中的元素可以是任意类型
字典	dict	{1：'food'，2：'taste'，3：'import'}	所有元素放在一对大括号中，元素之间使用逗号分隔，元素形式为"键：值"
元组	tuple	(2，-5，6)，(3，)	所有元素放在一对圆括号中，元素之间使用逗号分隔，如果元组中只有一个元素的话，后面的逗号不能省略
集合	set frozenset	{'a'，'b'，'c'}	所有元素放在一对大括号中，元素之间使用逗号分隔，元素不允许重复；另外，set 是可变的，而 frozenset 是不可变的
布尔型	bool	True，False	逻辑值，关系运算符、成员测试运算符、同一性测试运算符组成的表达式的值一般为 True 或 False
空类型	NoneType	None	空值



续表

对象类型	类型名称	示　　例	简 要 说 明
异常	Exception ValueError TypeError ⋮		Python 内置大量异常类，分别对应不同类型的异常
文件		f = open('data. dat', 'rb')	open 是 Python 的内置函数，使用指定的模式打开文件，返回文件对象
其他迭代对象		生成器对象、range 对象、zip 对象、enumerate 对象、map 对象、filter 对象等	具有惰性求值的特点
编程单元		函数(使用 def 定义) 类(使用 class 定义) 模块(类型为 module)	类和函数都属于可调用对象，模块用来集中存放函数、类、常量或其他对象

2.1.1　常量与变量

在表 2-1 中，第 3 列的示例除了最后 4 行之外，其他都是合法的 Python 常量。所谓常量，一般是指不需要改变也不能改变的字面值，例如一个数字 3，又例如一个列表[1,2,3]，都是常量。与常量相反，变量的值是可以变化的，这一点在 Python 中更是体现得淋漓尽致。在 Python 中，不需要事先声明变量名及其类型，直接赋值即可创建任意类型的对象变量。不仅变量的值是可以变化的，变量的类型也是随时可以发生改变的。例如，下面第一条语句创建了整型变量 x，并赋值为 3。

```
>>>x=3                          #整型变量
>>>type(x)                      #内置函数 type()用来查看变量类型
<class 'int'>
>>>type(x)==int
True
>>>isinstance(x,int)            #内置函数 isinstance()用来测试变量是否为指定类型
True
```

下面的语句创建了字符串变量 x，并赋值为'Hello world.'，之前的整型变量 x 不复存在。

```
>>>x='Hello world.'             #字符串变量
```

下面的语句创建了列表对象 x，并赋值为[1, 2, 3]，之前的字符串变量 x 也就不再存在了。这一点同样适用于元组、字典、集合和其他 Python 任意类型的对象，包括自定义类型的对象。

```
>>>x=[1,2,3]
```

Python 采用基于值的内存管理模式。赋值语句的执行过程是：首先把等号右侧表达

式的值计算出来,然后在内存中寻找一个位置把值存放进去,最后创建变量并指向这个内存地址。Python 中的变量并不直接存储值,而是存储了值的内存地址或者引用,这也是变量类型随时可以改变的原因。

虽然不需要在使用之前显式地声明变量及其类型,但 Python 是一种不折不扣的强类型编程语言,Python 解释器会根据赋值运算符右侧表达式的值来自动推断变量类型。其工作方式类似于“状态机”,变量被创建以后,除非显式修改变量类型或删除变量,否则变量将一直保持之前的类型。

如果变量出现在赋值运算符或复合赋值运算符(例如 += 、*= 等)的左边则表示创建变量或修改变量的值,否则表示引用该变量的值,这一点同样适用于使用下标来访问列表、字典等可变序列以及自定义对象中元素的情况。例如:

```
>>>x=3                           #创建整型变量
>>>print(x**2)                   #访问变量的值
9
>>>x+=6                          #修改变量的值
>>>x=[1,2,3]                     #创建列表对象
>>>x[1]=5                        #修改列表元素值
>>>print(x)                      #输出显示整个列表
[1,5,3]
>>>print(x[2])                   #输出显示列表指定元素
3
```

在 Python 中定义变量名的时候,需要注意以下问题。

(1) 变量名**必须**以字母或下画线开头,但以下画线开头的变量在 Python 中有特殊含义,请参考第 6 章内容。

(2) 变量名中**不能**有空格或标点符号(括号、引号、逗号、斜线、反斜线、冒号、句号、问号等)。

(3) **不能**使用关键字作为变量名,Python 关键字的介绍请见 2.3 节。要注意的是,随着 Python 版本的变化,关键字列表可能会有所变化。

(4) **不建议**使用系统内置的模块名、类型名或函数名以及已导入的模块名及其成员名作为变量名,这会改变其类型和含义,甚至会导致其他代码无法正常执行。可以通过 dir(__builtins__)查看所有内置对象名称。

(5) 变量名对英文字母的**大小写敏感**,例如 student 和 Student 是不同的变量。

2.1.2 数 字

在 Python 中,内置的数字类型有整数、实数和复数,借助于标准库 fractions 中的 Fraction 对象可以实现分数及其运算,而 fractions 中的 Decimal 类则实现了更高精度的运算。

1. 内置的整数、实数与复数

Python 支持任意大的数字,具体可以大到什么程度仅受内存大小的限制。由于精度



的问题，对于实数运算可能会有一定的误差，应尽量避免在实数之间直接进行相等性测试，而是应该以两者之差的绝对值是否足够小作为两个实数是否相等的依据。在数字的算术运算表达式求值时会进行隐式的类型转换，如果存在复数则都变成复数，如果没有复数但是有实数就都变成实数，如果都是整数则不进行类型转换。

```
>>> 9999 ** 99                        #这里**是幂乘运算符，等价于内置函数 pow()
990148353526723487602263124753282625570559528895791057324326529121794837894053
5134644221768269164339325869243866777662440320016237568214004329750512088202049
8009873555270384136230466997051069124380021820284037432937880069492030979195418
5117798434329591212159106298699938669908067573374724331208942425544893910910073
2050490316567892208895607329629262263058657065935949178962767563968485149009899
99
>>> 0.3 + 0.2                         #实数相加
0.5
>>> 0.4 - 0.1                         #实数相减，结果稍微有点偏差
0.30000000000000004
>>> 0.4 - 0.1 == 0.3                  #应尽量避免直接比较两个实数是否相等
False
>>> abs(0.4 - 0.1 - 0.3) < 1e-6       #这里 1e-6 表示 10 的 -6 次方
True
```

Python 内置支持复数类型及其运算，并且形式与数学上的复数完全一致。例如：

```
>>> x = 3 + 4j                        #使用 j 或 J 表示复数虚部
>>> y = 5 + 6j
>>> x + y                             #支持复数之间的加、减、乘、除以及幂乘等运算
(8+10j)
>>> x * y
(-9+38j)
>>> abs(x)                            #内置函数 abs()可用来计算复数的模
5.0
>>> x.imag                            #虚部
4.0
>>> x.real                            #实部
3.0
>>> x.conjugate()                     #共轭复数
(3-4j)
```

Python 3.6.x 支持**在数字中间位置使用单个下画线作为分隔来提高数字的可读性**，类似于数学上使用逗号作为千位分隔符。在 Python 数字中单个下画线可以出现在中间任意位置，但不能出现在开头和结尾位置，也不能使用多个连续的下画线。

```
>>> 1_000_000
1000000
>>> 1_2_3_4
1234
```



```
>>>1_2 +3_4j
(12 +34j)
>>>1_2.3_45
12.345
```

2. 分数

Python 标准库 fractions 中的 Fraction 对象支持分数运算，还提供了用于计算最大公约数的 gcd( ) 函数和高精度实数类 Decimal，这里重点介绍 Fraction 对象。

```
>>>from fractions import Fraction
>>>x =Fraction(3,5)                    #创建分数对象
>>>y =Fraction(3,7)
>>>x
Fraction(3,5)
>>>x ** 2                              #幂运算
Fraction(9,25)
>>>x.numerator                         #查看分子
3
>>>x.denominator                       #查看分母
5
>>>x +y                                #支持分数之间的四则运算，自动进行通分
Fraction(36,35)
>>>x -y
Fraction(6,35)
>>>x * y
Fraction(9,35)
>>>x/y
Fraction(7,5)
>>>x * 2                               #分数与数字之间的运算
Fraction(6,5)
>>>Fraction(3.5)                       #把实数转换为分数
Fraction(7,2)
```

3. 高精度实数

标准库 fractions 和 decimal 中提供的 Decimal 类实现了更高精度的运算。

```
>>>from fractions import Decimal
>>>1/9                                 #内置的实数类型
0.1111111111111111
>>>Decimal(1/9)                        #高精度实数
Decimal('0.111111111111111104943205418749130330979824066162109375')
>>>1/3
0.3333333333333333
```




```
>>>Decimal(1/3)
Decimal('0.333333333333333314829616256247390992939472198486328125')
>>>Decimal(1/9) +Decimal(1/3)
Decimal('0.4444444444444444197728216750')
```

从 Python 3.6.x 开始，Decimal 对象提供了一个新的方法 as_integer_ratio()，可以把实数转换成两个整数，这两个整数的商恰好等于该实数。

```
>>>from decimal import Decimal
>>>Decimal('-3.14').as_integer_ratio()
(-157,50)
```

2.1.3　字符串

在 Python 中，没有字符常量和变量的概念，只有字符串类型的常量和变量，单个字符也是字符串。使用单引号、双引号、三单引号、三双引号作为定界符(delimiter)来表示字符串，并且不同的定界符之间可以互相嵌套。Python 3.x 全面支持中文，中文和英文字母都作为一个字符对待，甚至可以使用中文作为变量名。除了支持使用加号运算符连接字符串以外，Python 字符串还提供了大量的方法支持查找、替换、排版等操作。很多内置函数和标准库对象也支持对字符串的操作，将在第 7 章进行详细介绍。这里先简单介绍一下字符串对象的创建和连接。

```
>>>x='Hello world.'                      #使用单引号作为定界符
>>>x="Python is a great language."       #使用双引号作为定界符
>>>x='''Tom said,"Let's go."'''          #不同定界符之间可以互相嵌套
>>>print(x)
Tom said,"Let's go."
>>>x='good '+'morning'                   #连接字符串
>>>x
'good morning'
>>>x='good ''morning'                    #连接字符串,仅适用于字符串常量
>>>x
'good morning'
>>>x='good '
>>>x=x'morning'                          #不适用于字符串变量
SyntaxError: invalid syntax
>>>x=x+'morning'                         #字符串变量之间的连接可以使用加号
>>>x
'good morning'
```

Python 3.x 除了支持 Unicode 编码的 str 类型字符串之外，还支持字节串类型 bytes。对 str 类型的字符串调用其 encode()方法进行编码得到 bytes 字节串，对 bytes 字节串调用其 decode()方法并指定正确的编码格式则得到 str 字符串。例如：

```
>>>type('Hello world')                   #默认字符串类型为 str
```

```
 <class 'str'>
 >>>type(b'Hello world')                  #在定界符前加上字母 b 表示字节串
 <class 'bytes'>
 >>>'Hello world'.encode('utf-8')         #使用 UTF-8 编码格式进行编码
b'Hello world'
 >>>'Hello world'.encode('gbk')           #使用 gbk 编码格式进行编码
b'Hello world'
 >>>'董付国'.encode('utf-8')               #对中文进行编码
b'\xe8\x91\xa3\xe4\xbb\x98\xe5\x9b\xbd'
 >>>_.decode('utf-8')                     #一个下画线表示最后一次正确输出结果
'董付国'
 >>>'董付国'.encode('gbk')
b'\xb6\xad\xb8\xb6\xb9\xfa'
 >>>_.decode('gbk')                       #对 bytes 字节串进行解码
'董付国'
```

2.1.4 列表、元组、字典、集合

列表、元组、字典和集合是 Python 中常用的序列类型,很多复杂的业务逻辑最终还是由这些基本数据类型来实现。表 2-2 比较了这几种结构的区别。

表 2-2 列表、元组、字典、集合的对比

比较项	列 表	元 组	字 典	集 合
类型名称	list	tuple	dict	set
定界符	方括号[ ]	圆括号( )	大括号{ }	大括号{ }
是否可变	是	否	是	是
是否有序	是	是	否	否
是否支持下标	是(使用序号作为下标)	是(使用序号作为下标)	是(使用"键"作为下标)	否
元素分隔符	逗号	逗号	逗号	逗号
对元素形式的要求	无	无	键:值	必须可哈希
对元素值的要求	无	无	"键"必须可哈希	必须可哈希
元素是否可重复	是	是	"键"不允许重复,"值"可以重复	否
元素查找速度	非常慢	很慢	非常快	非常快
新增和删除元素速度	尾部操作快,其他位置慢	不允许	快	快

下面的代码简单演示了这几种对象的创建与使用,更详细的介绍请参考第 3 章。

```
 >>>x_list=[1,2,3]                        #创建列表对象
 >>>x_tuple=(1,2,3)                       #创建元组对象
```



```
>>>x_dict={'a':97,'b':98,'c':99}          #创建字典对象
>>>x_set={1,2,3}                          #创建集合对象
>>>print(x_list[1])                       #使用下标访问指定位置的元素
2
>>>print(x_tuple[1])                      #元组也支持使用序号作为下标
2
>>>print(x_dict['a'])                     #字典对象的下标是“键”
97
>>>3 in x_set                             #成员测试
True
```

除了这几种结构之外，前面介绍的字符串也具有相似的操作，详见第 7 章。另外，Python 还提供了 range、map、zip、filter、enumerate、reversed 等大量迭代对象（迭代对象可以理解为表示数据流的对象，每次返回一个数据）。这些迭代对象大多具有与 Python 序列相似的操作方法，比较大的区别在于这些迭代对象大多具有惰性求值的特点，仅在需要时才给出新的元素，减少了对内存的占用，详见 2.4 节。

2.2　Python 运算符与表达式

Python 是面向对象的编程语言，在 Python 中一切都是对象。对象由数据和行为两部分组成，而行为主要通过方法来实现，通过一些特殊方法的重写，可以实现运算符重载。运算符也是表现对象行为的一种形式，不同类的对象支持的运算符有所不同，同一种运算符作用于不同的对象时也可能会表现出不同的行为，这正是“多态”的体现。

在 Python 中，单个常量或变量可以看作最简单的表达式，使用除赋值运算符之外的其他任意运算符和函数调用连接的式子也属于表达式。

除了算术运算符、关系运算符、逻辑运算符以及位运算符等常见运算符之外，Python 还支持一些特有的运算符，例如成员测试运算符、集合运算符、同一性测试运算符等。使用时需注意，Python 很多运算符具有多种不同的含义，作用于不同对象时的含义并不完全相同，非常灵活。常用的 Python 运算符如表 2-3 所示，运算符优先级遵循的规则为：算术运算符的优先级最高，其次是位运算符、成员测试运算符、关系运算符、逻辑运算符等，算术运算符遵循“先乘除，后加减”的基本运算原则。虽然 Python 运算符有一套严格的优先级规则，但是强烈建议在编写复杂表达式时使用圆括号说明其中的逻辑来提高代码可读性。记住，**圆括号是明确和改变表达式运算顺序的利器**，在适当的位置使用括号可以使得表达式的含义更加明确。

表 2-3　Python 运算符

运　算　符	功 能 说 明
+	算术加法，列表、元组、字符串合并与连接，正号
-	算术减法，集合差集，相反数



续表

运　算　符	功 能 说 明
*	算术乘法,序列重复
/	真除法
//	求整商,但如果操作数中有实数的话,结果为实数形式的整数
%	求余数,字符串格式化
**	幂运算
<、<=、>、>=、==、!=	(值)大小比较,集合的包含关系比较
or	逻辑或
and	逻辑与
not	逻辑非
in	成员测试
is	对象同一性测试,即测试是否为同一个对象或内存地址是否相同
|、^、&、<<、>>、~	位或、位异或、位与、左移位、右移位、位求反
&、|、^	集合交集、并集、对称差集
@	矩阵相乘运算符

221　算术运算符

(1) +运算符除了用于算术加法以外,还可以用于列表、元组、字符串的连接,但不支持不同类型的对象之间相加或连接。

```
>>>[1,2,3] +[4,5,6]                  #连接两个列表
[1,2,3,4,5,6]
>>>(1,2,3) +(4,)                     #连接两个元组
(1,2,3,4)
>>>'abcd' +'1234'                    #连接两个字符串
'abcd1234'
>>>'A' +1                            #不支持字符与数字相加,抛出异常
TypeError: Can't convert 'int' object to str implicitly
>>>True +3                           #Python 内部把 True 当作 1 处理
4
>>>False +3                          #把 False 当作 0 处理
3
```

(2) *运算符除了表示算术乘法,还可用于列表、元组、字符串这几个序列类型与整数的乘法,表示序列元素的重复,生成新的序列对象。字典和集合不支持与整数的相乘,因为其中的元素是不允许重复的。



```
>>>True * 3
3
>>>False * 3
0
>>>[1,2,3] * 3
[1,2,3,1,2,3,1,2,3]
>>>(1,2,3) * 3
(1,2,3,1,2,3,1,2,3)
>>>'abc' * 3
'abcabcabc'
```

(3) 运算符/和//在 Python 中分别表示算术除法和算术求整商(floor division)。

```
>>>3/2                    #数学意义上的除法
1.5
>>>15 // 4                #如果两个操作数都是整数,结果为整数
3
>>>15.0 // 4              #如果操作数中有实数,结果为实数形式的整数值
3.0
>>> -15//4                #向下取整
-4
```

(4) % 运算符可以用于整数或实数的求余数运算,还可以用于字符串格式化,但是并不推荐这种用法,关于字符串格式化的详细介绍请参考第 7 章。

```
>>>789 % 23               #余数
7
>>>123.45 % 3.2           #可以对实数进行余数运算,注意精度问题
1.849999999999996
>>>'%c,%d' % (65,65)      #把 65 分别格式化为字符和整数
'A,65'
>>>'%f,%s' % (65,65)      #把 65 分别格式化为实数和字符串
'65.000000,65'
```

(5) **运算符表示幂乘,等价于内置函数 pow()。例如:

```
>>>3 **2                  #3 的 2 次方,等价于 pow(3,2)
9
>>>pow(3,2,8)             #等价于 (3 **2)%8
1
>>>9 **0.5                #9 的 0.5 次方,即 9 的平方根
3.0
>>>(-9) **0.5             #可以计算负数的平方根
(1.8369701987210297e-16+3j)
```

2.2.2 关系运算符

Python 关系运算符最大的特点是可以连用,其含义与我们日常的理解完全一致。使

用关系运算符的一个最重要的前提是,操作数之间必须可比较大小。例如,把一个字符串和一个数字进行大小比较是毫无意义的,所以 Python 也不支持这样的运算。

```
>>>1 <3 <5                     #等价于 1 <3 and 3 <5
True
>>>3 <5 >2
True
>>>1 >6 <8
False
>>>1 >6 <math.sqrt(9)          #具有惰性求值或者逻辑短路的特点
False
>>>1 <6 <math.sqrt(9)          #还没有导入 math 模块,抛出异常
NameError: name 'math' is not defined
>>>import math
>>>1 <6 <math.sqrt(9)
False
>>>'Hello' >'world'            #比较字符串的大小
False
>>>[1,2,3] <[1,2,4]            #比较列表的大小
True
>>>'Hello' >3                  #字符串和数字不能比较
TypeError: unorderable types: str() >int()
>>>{1,2,3} <{1,2,3,4}          #测试是否子集
True
>>>{1,2,3} =={3,2,1}           #测试两个集合是否相等
True
>>>{1,2,4} >{1,2,3}            #集合之间的包含测试
False
>>>{1,2,4} <{1,2,3}
False
>>>{1,2,4} =={1,2,3}
False
```

2.2.3 成员测试运算符 in 与同一性测试运算符 is

成员测试运算符 in 用于成员测试,即测试一个对象是否为另一个对象的元素。

```
>>>3 in [1,2,3]                #测试 3 是否存在于列表 [1,2,3]中
True
>>>5 in range(1,10,1)          #range()是用来生成指定范围数字的内置函数
True
>>>'abc' in 'abcdefg'          #子字符串测试
True
>>>for i in (3,5,7):           #循环,成员遍历
```

```
    print(i,end='\t')
3   5   7
```

同一性测试运算符(identity comparison)is 用来测试两个对象是否是同一个，如果是则返回 True，否则返回 False。如果两个对象是同一个，两者具有相同的内存地址。

```
>>>3 is 3
True
>>>x=[300,300,300]
>>>x[0] is x[1]                  #基于值的内存管理,同一个值在内存中只有一份
True
>>>x=[1,2,3]
>>>y=[1,2,3]
>>>x is y                        #上面形式创建的 x和 y不是同一个列表对象
False
>>>x[0] is y[0]
True
>>>x.append(4)                   #不影响列表 y的值
>>>x
[1,2,3,4]
>>>y
[1,2,3]
>>>x=y                           #x 和 y指向同一个对象
>>>x is y
True
>>>x.append(4)                   #对 x进行操作会对 y造成同样的影响
>>>x
[1,2,3,4]
>>>y
[1,2,3,4]
```

2.2.4　位运算符与集合运算符

位运算符只能用于整数，其内部执行过程为：首先将整数转换为二进制数，然后右对齐，必要的时候左侧补 0，按位进行运算，最后再把计算结果转换为十进制数字返回。位与运算规则为 1&1 =1、1&0 =0&1 =0&0 =0，位或运算规则为 1|1 =1|0 =0|1 =1、0|0 =0，位异或运算规则为 1^1 =0^0 =0、1^0 =0^1 =1。另外，左移位时右侧补 0，每左移一位相当于乘以 2；右移位时左侧补 0，每右移一位相当于整除以 2。

```
>>>3<<2                          #把 3左移 2位
12
>>>3 & 7                         #位与运算
3
>>>3 | 8                         #位或运算
```

```
11
>>>3 ^ 5                              #位异或运算
6
```

集合的交集、并集、对称差集等运算借助于位运算符来实现，而差集则使用减号运算符实现（注意，并集运算符不是加号）。

```
>>>{1,2,3} | {3,4,5}                  #并集,自动去除重复元素
{1,2,3,4,5}
>>>{1,2,3} & {3,4,5}                  #交集
{3}
>>>{1,2,3} ^ {3,4,5}                  #对称差集
{1,2,4,5}
>>>{1,2,3} -{3,4,5}                   #差集
{1,2}
```

225 逻辑运算符

逻辑运算符 and、or、not 常用来连接条件表达式构成更加复杂的条件表达式，并且 and 和 or 具有惰性求值或逻辑短路的特点，当连接多个表达式时只计算必须要计算的值。例如，表达式 exp1 and exp2 等价于 exp1 if not exp1 else exp2，而表达式 exp1 or exp2 则等价于 exp1 if exp1 else exp2。在编写复杂条件表达式时充分利用这个特点，合理安排不同条件的先后顺序，在一定程度上可以提高代码的运行速度。另外要注意的是，运算符 and 和 or 并不一定会返回 True 或 False，而是得到最后一个被计算的表达式的值，但是运算符 not 一定会返回 True 或 False。

```
>>>3 >5 and a >3                      #注意,此时并没有定义变量 a
False
>>>3 >5 or a >3                       #3 >5 的值为 False,所以需要计算后面的表达式
NameError: name 'a' is not defined
>>>3 <5 or a >3                       #3 <5 的值为 True,不需要计算后面的表达式
True
>>>3 and 5                            #最后一个计算的表达式的值作为整个表达式的值
5
>>>3 and 5 >2
True
>>>3 not in [1,2,3]                   #逻辑非运算 not
False
>>>3 is not 5                         #not 的计算结果只能是 True 或 False 之一
True
>>>not 3
False
>>>not 0
True
```



2.2.6 矩阵乘法运算符@

从 Python 3.5 开始增加了一个新的矩阵相乘运算符@，不过由于 Python 没有内置的矩阵类型，所以该运算符常与扩展库 numpy 一起使用。另外，@ 符号还可以用来表示修饰器的用法，详见 5.1.2 节。

```
>>>import numpy                  #numpy是用于科学计算的 Python 扩展库
>>>x=numpy.ones(3)               #ones()函数用于生成全 1 矩阵,参数表示矩阵大小
>>>m=numpy.eye(3) * 3            #eye()函数用于生成单位矩阵
>>>m[0,2]=5                      #设置矩阵指定位置上元素的值
>>>m[2,0]=3
>>>x@m                           #矩阵相乘
array([ 6., 3., 8.])
```

2.2.7 补充说明

除了表 2-3 中列出的运算符之外，Python 还有赋值运算符 = 和 + =、- =、* =、/ =、// =、* * =、| =、^ = 等大量复合赋值运算符。例如，x + = 3 在语法上等价（注意，在功能的细节上可能会稍有区别，请参考 3.1.4 节）于 x = x + 3，其他复合赋值运算符与此类似，不再赘述。

Python 不支持 + + 和 - - 运算符，虽然在形式上有时候似乎可以这样用，但实际上是另外的含义，要注意和其他语言的区别。

```
>>>i=3
>>>++i                           #正正得正
3
>>>+(+3)                         #与++i等价
3
>>>i++                           #Python 不支持++运算符,语法错误
SyntaxError: invalid syntax
>>>--i                           #负负得正,等价于 -(-i)
3
>>>---i                          #等价于 -(-(-i))
-3
>>>i--                           #Python 不支持--运算符,语法错误
SyntaxError: invalid syntax
>>>-----(3+5)
-8
>>>3--5                          #等价于 3-(-5)
8
>>>3+-5                          #等价于 3+(-5)
-2
>>>3-+5
-2
```

2.3　Python 关键字简要说明

Python 关键字只允许用来表达特定的语义,不允许通过任何方式改变它们的含义,也不能用来作为变量名、函数名或类名等标识符。在 Python 开发环境中导入模块 keyword 之后,可以使用 print(keyword.kwlist)查看所有关键字,含义如表 2-4 所示。

表 2-4　Python 关键字含义

关键字	含　义
False	常量,逻辑假
None	常量,空值
True	常量,逻辑真
and	逻辑与运算
as	在 import 或 except 语句中给对象起别名
assert	断言,用来确认某个条件必须满足,可用来帮助调试程序
break	用在循环中,提前结束 break 所在层次的循环
class	用来定义类
continue	用在循环中,提前结束本次循环
def	用来定义函数
del	用来删除对象或对象成员
elif	用在选择结构中,表示 else if 的意思
else	可以用在选择结构、循环结构和异常处理结构中
except	用在异常处理结构中,用来捕获特定类型的异常
finally	用在异常处理结构中,用来表示不论是否发生异常都会执行的代码
for	构造 for 循环,用来迭代序列或可迭代对象中的所有元素
from	明确指定从哪个模块中导入什么对象,例如 from math import sin;还可以与 yield 一起构成 yield 表达式
global	定义或声明全局变量
if	用在选择结构中
import	用来导入模块或模块中的对象
in	成员测试
is	同一性测试
lambda	用来定义 lambda 表达式,类似于函数
nonlocal	用来声明 nonlocal 变量

续表

关键字	含　　义
not	逻辑非运算
or	逻辑或运算
pass	空语句，执行该语句时什么都不做，常用作占位符
raise	用来显式抛出异常
return	在函数中用来返回值，如果没有指定返回值，表示返回空值 None
try	在异常处理结构中用来限定可能会引发异常的代码块
while	用来构造 while 循环结构，只要条件表达式等价于 True 就重复执行限定的代码块
with	上下文管理，具有自动管理资源的功能
yield	在生成器函数中用来返回值

2.4　Python 常用内置函数用法精要

内置函数(built-in functions，BIF)是 Python 内置对象类型之一，不需要额外导入任何模块即可直接使用，这些内置对象都封装在内置模块__builtins__之中，用 C 语言实现并且进行了大量优化，具有非常快的运行速度，推荐优先使用。使用内置函数 dir() 可以查看所有内置函数和内置对象：

```
>>>dir(__builtins__)
```

使用 help(函数名)可以查看某个函数的用法。另外，也可以不导入模块而直接使用 help(模块名)查看该模块的帮助文档，例如 help('math')。常用的内置函数及其功能简要说明如表 2-5 所示，其中方括号内的参数可以省略。

表 2-5　Python 常用内置函数

函　　数	功能简要说明
abs(x)	返回数字 x 的绝对值或复数 x 的模
all(iterable)	如果可迭代对象 iterable 中所有元素 x 都等价于 True，也就是对于所有元素 x 都有 bool(x) 等于 True，则返回 True。对于空的可迭代对象也返回 True
any(iterable)	只要可迭代对象 iterable 中存在元素 x 使得 bool(x) 为 True，则返回 True。对于空的可迭代对象，返回 False
ascii(obj)	把对象转换为 ASCII 码表示形式，必要的时候使用转义字符来表示特定的字符
bin(x)	把整数 x 转换为二进制串表示形式
bool(x)	返回与 x 等价的布尔值 True 或 False

续表

函　　数	功能简要说明
bytes(x)	生成字节串,或把指定对象 x 转换为字节串表示形式
callable(obj)	测试对象 obj 是否可调用。类和函数是可调用的,包含__call__()方法的类的对象也是可调用的
compile()	用于把 Python 代码编译成可被 exec()或 eval()函数执行的代码对象
complex(real, [imag])	返回复数
chr(x)	返回 Unicode 编码为 x 的字符
delattr(obj, name)	删除属性,等价于 del obj. name
dir(obj)	返回指定对象或模块 obj 的成员列表,如果不带参数则返回当前作用域内所有标识符
divmod(x, y)	返回包含整商和余数的元组((x－x% y)/y, x% y)
enumerate (iterable [, start])	返回包含元素形式为(start, iterable[0]),(start＋1, iterable[1]),(start＋2, iterable[2]),…的迭代器对象,start 表示索引的起始值
eval (s [, globals [, locals]])	计算并返回字符串 s 中表达式的值
exec(x)	执行代码或代码对象 x
exit()	退出当前解释器环境
filter(func, seq)	返回 filter 对象,其中包含序列 seq 中使得单参数函数 func 返回值为 True 的那些元素,如果函数 func 为 None 则返回包含 seq 中等价于 True 的元素的 filter 对象
float(x)	把整数或字符串 x 转换为浮点数并返回
frozenset([x])	创建不可变的集合对象
getattr (obj, name [, default])	获取对象中指定属性的值,等价于 obj. name,如果不存在指定属性则返回 default 的值,如果要访问的属性不存在并且没有指定 default 则抛出异常
globals()	返回包含当前作用域内全局变量及其值的字典
hasattr(obj, name)	测试对象 obj 是否具有名为 name 的成员
hash(x)	返回对象 x 的哈希值,如果 x 不可哈希则抛出异常
help(obj)	返回对象 obj 的帮助信息
hex(x)	把整数 x 转换为十六进制串
id(obj)	返回对象 obj 的标识(内存地址)
input([提示])	显示提示,接收键盘输入的内容,返回字符串
int(x[, d])	返回实数(float)、分数(Fraction)或高精度实数(Decimal)x 的整数部分,或把 d 进制的字符串 x 转换为十进制并返回,d 默认为十进制
isinstance (obj, class-or-type-or-tuple)	测试对象 obj 是否属于指定类型(如果有多个类型的话需要放到元组中)的实例



续表

函　　数	功能简要说明
issubclass(cls, class_or_tuple)	测试类 cls 是否为指定类型的子类，用法与 isinstance() 函数相似
iter(…)	返回指定对象的可迭代对象
len(obj)	返回对象 obj 包含的元素个数，适用于列表、元组、集合、字典、字符串以及 range 对象，不适用于具有惰性求值特点的生成器对象和 map、zip 等迭代对象
list([x])、set([x])、tuple([x])、dict([x])	把对象 x 转换为列表、集合、元组或字典并返回，或生成空列表、空集合、空元组、空字典
locals()	返回包含当前作用域内局部变量及其值的字典
map(func, *iterables)	返回包含若干函数值的 map 对象，函数 func 的参数分别来自于 iterables 指定的一个或多个迭代对象，
max(…)、min(…)	返回多个值中或者包含有限个元素的可迭代对象中所有元素的最大值、最小值，要求所有元素之间可比较大小，允许指定排序规则，参数为可迭代对象时还允许指定默认值
next(iterator[, default])	返回迭代对象 x 中的下一个元素，允许指定迭代结束之后继续迭代时返回的默认值
oct(x)	把整数 x 转换为八进制串
open(fn[, mode])	以指定模式 mode 打开文件 fn 并返回文件对象
ord(x)	返回 1 个字符 x 的 Unicode 编码
pow(x, y, z=None)	返回 x 的 y 次方，等价于 x ** y 或(x ** y) % z
print(value, …, sep=' ', end='\n', file=sys.stdout, flush=False)	基本输出函数，默认输出到屏幕，相邻数据使用空格分隔，以换行符结束所有数据的输出
quit()	退出当前解释器环境
range([start,] end [, step])	返回 range 对象，其中包含左闭右开区间[start,end)内以 step 为步长的整数
reduce(func, sequence[, initial])	将双参数的函数 func 以迭代的方式从左到右依次应用至序列 seq 中每个元素，并把中间计算结果作为下一次计算的操作数之一，最终返回单个值作为结果。在 Python 2.x 中该函数为内置函数，在 Python 3.x 中需要从 functools 中导入 reduce 函数再使用
repr(obj)	返回对象 obj 的规范化字符串表示形式，对于大多数对象有 eval(repr(obj)) == obj
reversed(seq)	返回 seq(可以是列表、元组、字符串、range 等对象)中所有元素逆序后的迭代器对象，不适用于具有惰性求值特点的生成器对象和 map、zip 等可迭代对象
round(x [, 小数位数])	对 x 进行四舍五入，若不指定小数位数，则返回整数
sorted(iterable, key=None, reverse=False)	返回排序后的列表，其中 iterable 表示要排序的序列或迭代对象，key 用来指定排序规则或依据，reverse 用来指定升序或降序



续表

函　　数	功能简要说明
str(obj)	把对象 obj 直接转换为字符串
sum(x, start = 0)	返回序列 x 中所有元素之和,允许指定起始值 start,返回 start + sum(x)
type(obj)	返回对象 obj 的类型
zip(seq1 [, seq2 [...]])	返回 zip 对象,其中元素为(seq1[i], seq2[i], ...)形式的元组,最终结果中包含的元素个数取决于所有参数序列或可迭代对象中最短的那个

内置函数数量众多且功能强大,很难一下子全部解释清楚,下面先简单介绍其中一部分,后面的章节中将根据内容组织的需要逐步展开和演示更多函数更加巧妙的用法。遇到不熟悉的函数可以通过内置函数 help()查看使用帮助。另外,在编写程序时应优先考虑使用内置函数,因为内置函数不仅成熟、稳定,而且速度相对较快。

2.4.1 类型转换与类型判断

(1) 内置函数 bin()、oct()、hex()用来将整数转换为二进制、八进制和十六进制形式,这 3 个函数都要求参数必须为整数。

```
>>> bin(555)                     #把数字转换为二进制串
'0b1000101011'
>>> oct(555)                     #转换为八进制串
'0o1053'
>>> hex(555)                     #转换为十六进制串
'0x22b'
```

内置函数 int()用来将其他形式的数字转换为整数,参数可以为整数、实数、分数或合法的数字字符串,当参数为数字字符串时,还允许指定第二个参数 base 用来说明数字字符串的进制。其中,base 的取值应为 0 或 2 ~ 36 之间的整数,其中 0 表示按数字字符串隐含的进制进行转换。

```
>>> int(-3.2)                    #把实数转换为整数
-3
>>> from fractions import Fraction, Decimal
>>> x = Fraction(7,3)
>>> x
Fraction(7,3)
>>> int(x)                       #把分数转换为整数
2
>>> x = Decimal(10/3)
>>> x
Decimal('3.333333333333333481363069950020872056484222412109375')
>>> int(x)                       #把高精度实数转换为整数
3
```




```
>>>int('0x22b',16)          #把十六进制数转换为十进制数
555
>>>int('22b',16)            #与上一行代码等价
555
>>>int(bin(54321),2)        #二进制与十进制之间的转换
54321
>>>int('0b111')             #非十进制字符串进,必须指定第二个参数
ValueError: invalid literal for int() with base 10: '0b111'
>>>int('0b111',0)           #第二个参数 0 表示使用字符串隐含的进制
7
>>>int('0b111',6)           #第二个参数应与隐含的进制一致
ValueError: invalid literal for int() with base 6: '0b111'
>>>int('0b111',2)
7
>>>int('111',6)             #字符串没有隐含进制
                            #第二个参数可以为 2~36 之间的数字
43
```

内置函数 float( )用来将其他类型数据转换为实数,complex( )可以用来生成复数。

```
>>>float(3)                 #把整数转换为实数
3.0
>>>float('3.5')             #把数字字符串转换为实数
3.5
>>>float('inf')             #无穷大,其中 inf 不区分大小写
inf
>>>complex(3)               #指定实部
(3+0j)
>>>complex(3,5)             #指定实部和虚部
(3+5j)
>>>complex('inf')           #无穷大
(inf+0j)
>>>float('nan')             #非数字,not a number 的缩写
nan
>>>complex('nan')
(nan+0j)
>>>nan=float('nan')
>>>nan==float('nan')        #无法比较大小
False
>>>nan >=float('nan')
False
>>>nan<=float('nan')
False
```

(2) ord( )和 chr( )是一对功能相反的函数,ord( )用来返回单个字符的 Unicode 码,



而 chr( )则用来返回 Unicode 编码对应的字符，str( )则直接将其任意类型参数转换为字符串。

```
>>>ord('a')                          #查看指定字符的 Unicode 编码
97
>>>chr(65)                           #返回数字 65 对应的字符
'A'
>>>chr(ord('A')+1)                   #Python 不允许字符串和数字之间的加法操作
'B'
>>>chr(ord('国')+1)                  #支持中文
'图'
>>>ord('董')                         #这个用法仅适用于 Python 3.x
33891
>>>ord('付')
20184
>>>ord('国')
22269
>>>''.join(map(chr,(33891,20184,22269)))
'董付国'
>>>str(1234)                         #直接变成字符串
'1234'
>>>str([1,2,3])
'[1,2,3]'
>>>str((1,2,3))
'(1,2,3)'
>>>str({1,2,3})
'{1,2,3}'
```

内置类 ascii 可以把对象转换为 ASCII 码表示形式，必要的时候使用转义字符来表示特定的字符。

```
>>>ascii('a')
"'a'"
>>>ascii('董付国')
"'\\u8463\\u4ed8\\u56fd'"
>>>eval(_)                           #对字符串进行求值
'董付国'
```

内置类 bytes 用来生成字节串，或者把指定对象转换为特定编码的字节串。

```
>>>bytes()                           #生成空字节串
b''
>>>bytes(3)                          #生成长度为 3 的字节串
b'\x00\x00\x00'
>>>bytes('董付国','utf-8')           #把字符串转换为字节串
b'\xe8\x91\xa3\xe4\xbb\x98\xe5\x9b\xbd'
```

```
>>>bytes('董付国','gbk')                    #可以指定不同的编码格式
b'\xb6\xad\xb8\xb6\xb9\xfa'
>>>str(_,'gbk')                             #使用同样的编码格式进行解码
'董付国'
>>>'董付国'.encode('gbk')                   #等价于使用 bytes()进行转换
b'\xb6\xad\xb8\xb6\xb9\xfa'
>>>_.decode('gbk')                          #等价于使用 str()进行转换
'董付国'
>>>x='董付国'.encode()
>>>list(x)                                  #把字节串转换为列表
[232,145,163,228,187,152,229,155,189]
>>>bytes(_)                                 #把整数列表转换为字节串
b'\xe8\x91\xa3\xe4\xbb\x98\xe5\x9b\xbd'
>>>_.decode()
'董付国'
```

（3）list()、tuple()、dict()、set()、frozenset()用来把其他类型的数据转换成为列表、元组、字典、可变集合和不可变集合，或者创建空列表、空元组、空字典和空集合。

```
>>>list(range(5))                           #把 range 对象转换为列表
[0,1,2,3,4]
>>>tuple(_)                                 #一个下画线表示上一次正确的输出结果
(0,1,2,3,4)
>>>dict(zip('1234','abcde'))                #创建字典
{'4': 'd','2': 'b','3': 'c','1': 'a'}
>>>set('1112234')                           #创建可变集合，自动去除重复
{'4','2','3','1'}
>>>_.add('5')
>>>_
{'2','1','3','4','5'}
>>>frozenset('1112234')                     #创建不可变集合，自动去除重复
frozenset({'2','1','3','4'})
>>>_.add('5')                               #不可变集合 frozenset 不支持元素添加与删除
AttributeError: 'frozenset' object has no attribute 'add'
```

实际上，这里的 list、tuple、dict、set、frozenset 以及前面刚刚介绍的 int、float、complex、bytes、str 都是 Python 的内置类。之所以可以把这些类像函数一样调用，是因为构造方法的存在。也就是说，上面的操作实际上是间接地调用了这些类的构造方法。另外，由于这些函数的调用会生成新的类型，也往往被称为“工厂函数”。

（4）内置函数 type()和 isinstance()可以用来判断数据类型，常用来对函数参数进行检查，可以避免错误的参数类型导致函数崩溃或返回意料之外的结果。不过，从另一方面来讲，过多的使用 type()和 isinstance()函数会在一定程度上影响多态，建议谨慎使用。

```
>>>type(3)                                  #查看 3 的类型
<class 'int'>
```

```
>>>type([3])                          #查看[3]的类型
<class 'list'>
>>>type({3}) in (list,tuple,dict)     #判断{3}是否为 list、tuple 或 dict 类型的实例
False
>>>type({3}) in (list,tuple,dict,set)
                                      #判断{3}是否为 list、tuple、dict 或 set 的实例
True
>>>isinstance(3,int)                  #判断 3 是否为 int 类型的实例
True
>>>isinstance(3j,int)
False
>>>isinstance(3j,(int,float,complex))
                                      #判断 3 是否为 int、float 或 complex 类型的实例
True
```

内置函数 callable( )用来测试对象是否可调用,Python 中函数和类都是可调用的,包含__call__( )方法的类的对象也是可调用的。另外,Python 标准库 inspect 提供了 isclass( )、ismethod( )、isgenerator( )、isroutine( )等大量函数用来测试对象类型,感兴趣的读者可以深入挖掘一些,或许会有意外惊喜。

2.4.2 最值与求和

max( )、min( )、sum( )这 3 个内置函数分别用于计算列表、元组或其他包含有限个元素的可迭代对象中所有元素最大值、最小值以及所有元素之和。sum( )默认(可以通过 start 参数来改变)支持包含数值型元素的序列或可迭代对象,max( )和 min( )则要求序列或可迭代对象中的元素之间可比较大小。

```
>>>from random import randint
>>>a=[randint(1,100) for i in range(10)]     #包含 10 个[1,100]之间随机数的列表
>>>print(max(a),min(a),sum(a))               #最大值、最小值、所有元素之和
>>>sum(a)/len(a)                             #平均值
```

函数 max( )和 min( )还支持 default 参数和 key 参数,其中 default 参数用来指定可迭代对象为空时默认返回的最大值或最小值,而 key 参数用来指定比较大小的依据或规则,可以是函数或 lambda 表达式。函数 sum( )还支持 start 参数,用来控制求和的初始值。

```
>>>max(['2','111'])                 #不指定排序规则
'2'
>>>max(['2','111'],key=len)         #返回最长的字符串
'111'
>>>print(max([],default=None))      #对空列表求最大值,返回空值 None
None
>>>from random import randint
>>>lst=[[randint(1,50) for i in range(5)] for j in range(30)]
                                    #列表推导式,生成包含 30 个子列表的列表
```

```
                                    #每个子列表中包含 5 个介于[1,50]区间的整数
>>>max(*lst,key=sum)                #返回元素之和最大的子列表,略去结果
>>>max(lst,key=sum)                 #与上面的代码等价,这是 max()的另一个用法
>>>max(lst,key=lambda x: x[1])      #所有子列表中第 2 个元素最大的子列表
>>>sum(range(1,11))                 #sum()函数的 start 参数默认为 0
55
>>>sum(range(1,11),5)               #指定 start 参数为 5,等价于 5+sum(range(1,11))
60
>>>sum([[1,2],[3],[4]],[])          #这个操作占用空间较大,慎用
[1,2,3,4]
>>>sum(2**i for i in range(200))    #等比数列前 n 项的和,1+2+4+8+… +2^199
1606938044258990275541962092341162602522202993782792835301375
>>>int('1'*200,2)                   #等价于上一行代码,但速度快很多
1606938044258990275541962092341162602522202993782792835301375
>>>int('1'*200,7)                   #比值 q 为 2~36 之间的整数时,都可以这样做
1743639715219059529169816601969468943303198091695038943325023347339187627904043
7086290637691515606750488442080420910523623438633906139318646917923778899694224 39576020000
>>>sum(range(101))                  #101 个人开会,互相握手次数,不重复握手
5050
>>>101*100 // 2                     #每个人都与其他所有人握手,但不重复握手
5050
```

如果用 help( ) 函数查看 sum( ) 函数的帮助文档时,会发现 sum( ) 函数的最后一个参数是斜线,实际上这个斜线并不是 sum( ) 函数的参数,只是用来表明这个函数只接收位置参数,而不允许以关键参数的形式进行传值,如果遇到其他函数或对象方法显示这样的帮助文档也表示同样的含义。这样的函数是用 C 开发的,并对参数传值形式做了要求,在 Python 中并不允许定义这样的函数。这涉及 Argument Clinic 的概念,感兴趣的朋友可以查阅有关资料。

```
>>>help(sum)                        #查看 sum()函数的帮助
Help on built-in function sum in module builtins:

sum(iterable,start=0,/)
    Return the sum of a 'start' value (default: 0) plus an iterable of numbers

    When the iterable is empty,return the start value.
    This function is intended specifically for use with numeric values and may
    reject non-numeric types.
>>>sum([1,2,3],4)                   #按位置参数对 start 进行传值
10
>>>sum([1,2,3],start=4)             #不允许使用关键参数,引发异常
TypeError: sum() takes no keyword arguments
>>>def demo(a,b,/):                 #在 Python 中不允许这样定义函数,语法错误
SyntaxError: invalid syntax
```

2.4.3 基本输入输出

input()和 print()是 Python 的基本输入输出函数,前者用来接收用户的键盘输入,后者用来把数据以指定的格式输出到标准控制台或指定的文件对象。不论用户输入什么内容,input()一律作为字符串对待,必要的时候可以使用内置函数 int()、float()或 eval()对用户输入的内容进行类型转换。

```
>>>x=input('Please input: ')
Please input: 345
>>>x
'345'
>>>type(x)                       #把用户的输入作为字符串对待
<class 'str'>
>>>int(x)                        #转换为整数
345
>>>eval(x)                       #对字符串求值,或类型转换
345
>>>x=input('Please input: ')
Please input: [1,2,3]
>>>x
'[1,2,3]'
>>>type(x)
<class 'str'>
>>>eval(x)
[1,2,3]
>>>x=input('Please input:')      #不论用户输入什么,都作为一个字符串来对待
Please input:'hello world'
>>>x                             #如果本来就想输入字符串,就不用再输入引号了
"'hello world'"
>>>eval(x)
'hello world'
```

内置函数 print()用于输出信息到标准控制台或指定文件,语法格式为

```
print(value1,value2,…,sep=' ',end='\n',file=sys.stdout,flush=False)
```

其中,sep 参数之前为需要输出的内容(可以有多个);sep 参数用于指定数据之间的分隔符,默认为空格;file 参数用于指定输出位置,默认为标准控制台,也可以重定向输出到文件。例如:

```
>>>print(1,3,5,7,sep='\t')       #修改默认分隔符
1	3	5	7
>>>for i in range(10):           #修改 end 参数,每个输出之后不换行
    print(i,end=' ')
0 1 2 3 4 5 6 7 8 9
```




```
>>>with open('test.txt','a+') as fp:
    print('Hello world! ',file=fp)#重定向,将内容输出到文件中
```

另外,Python 标准库 sys 还提供了 read( ) 和 readline( ) 方法用来从键盘接收指定数量的字符。例如:

```
>>>import sys
>>>x=sys.stdin.read(5)          #读取 5 个字符,如果不足 5 个,等待继续输入
asd
s
>>>x
'asd\ns'
>>>x=sys.stdin.read(5)          #读取 5 个字符,如果超出 5 个,截断
abcdefghijklmnop
>>>x
'abcde'
>>>x=sys.stdin.read(5)          #从缓冲区内继续读取 5 个字符
>>>x
'fghij'
>>>x=sys.stdin.read(5)
>>>x
'klmno'
>>>x=sys.stdin.read(5)          #缓冲区内不足 5 个字符,就等待用户继续输入
1234
>>>x
'p\n123'
>>>x=sys.stdin.readline()       #从缓冲区内读取字符,遇到换行符就结束
>>>x
'4\n'
>>>x=sys.stdin.readline()
abcd
>>>x
'abcd\n'
>>>x=sys.stdin.readline(13)     #如果缓冲区内容比需要的少,遇到换行符也结束
abcdefg
>>>x
'abcdefg\n'
>>>x=sys.stdin.readline(13)     #如果缓冲区内容比需要的多,就截断
abcdefghijklmnopqrst
>>>x
'abcdefghijklm'
>>>x=sys.stdin.readline(13)     #从缓冲区继续读取
>>>x
'nopqrst\n'
```

Python 标准库 pprint 则提供了更加友好的输出函数(pretty printer)pprint(),可以更好地控制输出格式,如果要输出的内容多于一行则会自动添加换行和缩进来更好地展示内容的结构。例如:

```
>>>import pprint
>>>t=[[[['black','cyan'],'white',['green','red']],[['magenta','yellow'],'blue']]]
>>>pprint.pprint(t)                  #默认 width=80
[[[['black','cyan'],'white',['green','red']],
  [['magenta','yellow'],'blue']]]
>>>pprint.pprint(t,width=50)
[[[['black','cyan'],'white',['green','red']],
  [['magenta','yellow'],'blue']]]
>>>pprint.pprint(t,width=30)         #根据需要进行换行和缩进
[[[['black','cyan'],
    'white',
    ['green','red']],
  [['magenta','yellow'],
    'blue']]]
```

2.4.4 排序与逆序

sorted()对列表、元组、字典、集合或其他可迭代对象进行排序并返回新列表,reversed()对可迭代对象(生成器对象和具有惰性求值特性的 zip、map、filter、enumerate 等类似对象除外)进行翻转(首尾交换)并返回可迭代的 reversed 对象。

```
>>>x=list(range(11))
>>>import random
>>>random.shuffle(x)                 #打乱顺序
>>>x
[2,4,0,6,10,7,8,3,9,1,5]
>>>sorted(x)                         #以默认规则排序
[0,1,2,3,4,5,6,7,8,9,10]
>>>sorted(x,key=lambda item:len(str(item)),reverse=True)
                                     #按转换成字符串以后的长度降序排列
[10,2,4,0,6,7,8,3,9,1,5]
>>>sorted(x,key=str)                 #按转换成字符串以后的大小升序排列
[0,1,10,2,3,4,5,6,7,8,9]
>>>x                                 #不影响原来列表的元素顺序
[2,4,0,6,10,7,8,3,9,1,5]
>>>x=['aaaa','bc','d','b','ba']
>>>sorted(x,key=lambda item: (len(item),item))
                                                        #先按长度排序,长度一样的正常排序
['b','d','ba','bc','aaaa']
>>>reversed(x)                                          #逆序,返回 reversed对象
```

```
<list_reverseiterator object at 0x0000000003089E48>
>>>list(reversed(x))                          #reversed对象是可迭代的
[5,1,9,3,8,7,10,6,0,4,2]
```

2.4.5　枚举与迭代

enumerate()函数用来枚举可迭代对象中的元素，返回可迭代的 enumerate 对象，其中每个元素都是包含索引和值的元组。

```
>>>list(enumerate('abcd'))                     #枚举字符串中的元素
[(0,'a'),(1,'b'),(2,'c'),(3,'d')]
>>>list(enumerate(['Python','Greate']))        #枚举列表中的元素
[(0,'Python'),(1,'Greate')]
>>>list(enumerate({'a':97,'b':98,'c':99}.items()))   #枚举字典中的元素
[(0,('c',99)),(1,('a',97)),(2,('b',98))]
>>>for index,value in enumerate(range(10,15)):   #枚举 range对象中的元素
    print((index,value),end=' ')
(0,10) (1,11) (2,12) (3,13) (4,14)
```

内置函数 enumerate()还支持一个 start 参数，用来指定枚举时的索引起始值。

```
>>>for item in enumerate(range(5),6):          #索引从 6开始
    print(item,end=',')
(6,0),(7,1),(8,2),(9,3),(10,4),
```

iter()函数用来返回指定对象的迭代器，有两种用法：iter(iterable)和 iter(callable, sentinel)，前者要求参数必须为序列或者有自己的迭代器，后者会持续调用参数 callable 直至其返回 sentinel。next()函数用来返回可迭代对象中的下一个元素，适用于生成器对象以及 zip、enumerate、reversed、map、filter、iter 等对象，等价于这些对象的__next__()方法。

```
>>>x=[1,2,3]
>>>next(x)
TypeError: 'list' object is not an iterator
>>>y=iter(x)                                   #根据列表创建迭代器对象
>>>next(y)
1
>>>next(y)
2
>>>x=range(1,100,3)
>>>next(x)                                 #range对象不是迭代器对象
TypeError: 'range' object is not an iterator
>>>x=iter(x)                               #把 range对象转换为迭代器对象
>>>next(x)
1
```

```
>>>next(x)
4
>>>x={1,2,3}
>>>y=iter(x)                                #根据集合创建迭代器对象
>>>next(y)
1
>>>class T:
  def __init__(self,seq):
    self.__data=list(seq)
  def __iter__(self):                       #特殊方法,对应于内置函数 iter()
    return iter(self.__data)
>>>t=T(range(3))
>>>next(t)                                  #对象 t 不可迭代
TypeError: 'T' object is not an iterator
>>>ti=iter(t)                               #根据 t 创建迭代器对象
>>>next(ti)
0
>>>next(ti)
1
>>>from queue import Queue
>>>q=Queue()                                #创建队列对象
>>>for i in range(5):
    q.put(i)                                #依次放入 5 个数字
>>>q.put('END')                             #放入结束标志
>>>def test():
    return q.get()
>>>for item in iter(test,'END'):            #持续执行 test()函数,直到返回'END'
    print(item,end=' ')
0 1 2 3 4
>>>x=map(int,'1234')                        #map 对象支持 next()函数
>>>next(x)
1
>>>x.__next__()                             #特殊方法,等价于 next(x)
2
>>>x=reversed('12345678')                   #reversed 对象支持 next()函数
>>>for i in range(4):
    print(next(x),end=' ')
8 7 6 5
>>>x=enumerate({'a':97,'b':98,'c':99}.items())
                                            #enumerate 对象支持 next()函数
>>>next(x)
(0,('c',99))
```

2.4.6　map()、reduce()、filter()

map()、reduce()、filter()是 Python 中很常用的几个函数，也是 Python 支持函数式编程的重要体现。要注意的是，在 Python 3.x 中，reduce()不是内置函数，而是放到了标准库 functools 中，需要先导入再使用。

函数式编程把问题分解为一系列的函数操作，输入依次流入和流出一系列函数，最终完成预定任务和目标。在理想状态下，每个函数只是接收输入并在简单处理后产生输出，这些输出完全取决于输入和函数指定的操作并且仅通过函数返回值来体现，而不会引入或造成任何副作用(例如，输入和一些内部状态共同决定输出、修改输入的数据、把一些数据和状态输出到屏幕或文件等)。函数式编程具有很多优点，例如，容易构建一个数学模型来证明程序的正确性，程序更加模块化，容易调试和测试，代码复用率高，等等。

内置函数 map()把一个函数 func 依次映射到序列或迭代器对象的每个元素上，并返回一个可迭代的 map 对象作为结果，map 对象中每个元素是原序列中元素经过函数 func 处理后的结果，map()函数不对原序列或迭代器对象做任何修改。

```
>>>list(map(str,range(5)))                          #把列表中元素转换为字符串
['0','1','2','3','4']
>>>def add5(v):                                     #单参数函数
    return v+5
>>>list(map(add5,range(10)))                        #把单参数函数映射到一个序列的所有
元素
[5,6,7,8,9,10,11,12,13,14]
>>>def add(x,y):                                    #可以接收 2 个参数的函数
    return x+y
>>>list(map(add,range(5),range(5,10)))              #把双参数函数映射到两个序列上
[5,7,9,11,13]
>>>list(map(lambda x,y: x+y,range(5),range(5,10)))
[5,7,9,11,13]
>>>def myMap(lst,value):                            #自定义函数
    return map(lambda item: item+value,lst)
>>>list(myMap(range(5),5))                          #每个数字加 5
[5,6,7,8,9]
>>>list(myMap(range(5),8))                          #每个数字加 8
[8,9,10,11,12]
>>>def myMap(iterable,op,value):                    #自定义函数
    if op not in '+-*/':                            #实现序列与数字的四则运算
        return 'Error operator'
    func=lambda i:eval(repr(i)+op+repr(value))
    return map(func,iterable)
>>>list(myMap(range(5),'+',5))
[5,6,7,8,9]
>>>list(myMap(range(5),'-',5))
```

```
[-5, -4, -3, -2, -1]
>>>list(myMap(range(5),'*',5))
[0,5,10,15,20]
>>>list(myMap(range(5),'/',5))
[0.0,0.2,0.4,0.6,0.8]
>>>import random
>>>x=random.randint(1,1e30)                     #生成指定范围内的随机整数
>>>x
839746558215897242220046223150
>>>list(map(int,str(x)))                        #提取大整数每位上的数字
[8,3,9,7,4,6,5,5,8,2,1,5,8,9,7,2,4,2,2,2,0,0,4,6,2,2,3,1,5,0]
```

标准库 functools 中的函数 reduce( ) 可以将一个接收 2 个参数的函数以迭代累积的方式从左到右依次作用到一个序列或迭代器对象的所有元素上，并且允许指定一个初始值。例如，reduce(lambda x, y: x + y, [1, 2, 3, 4, 5]) 计算过程为((((1 +2) +3) +4) +5)，第一次计算时 x 为 1 而 y 为 2，再次计算时 x 的值为(1 +2) 而 y 的值为 3，再次计算时 x 的值为((1 +2) +3) 而 y 的值为 4，以此类推，最终完成计算并返回((((1 +2) +3) +4) +5) 的值。

```
>>>from functools import reduce
>>>seq=list(range(1,10))                        #也可以不用转换为列表
>>>reduce(add,seq)                              #add 是上一段代码中定义的函数
45
>>>reduce(lambda x,y: x+y,seq)                  #使用 lambda 表达式实现相同功能
45
```

上面实现数字累加的代码运行过程如图 2-1 所示。

```
>>>import operator                              #标准库 operator 提供了大量运算
>>>operator.add(3,5)                            #可以像普通函数一样直接调用
8
>>>reduce(operator.add,seq)                     #使用 add 运算
45
>>>reduce(operator.add,seq,5)                   #指定累加的初始值为 5
50
>>>reduce(operator.mul,seq)                     #乘法运算
362880
>>>reduce(operator.mul,range(1,6))              #5 的阶乘
120
>>>reduce(operator.add,map(str,seq))            #转换成字符串再累加
'123456789'
>>>''.join(map(str,seq))                        #使用 join()方法实现字符串连接
'123456789'
>>>reduce(operator.add,[[1,2],[3]],[])          #这个操作占用空间较大，慎用
[1,2,3]
```

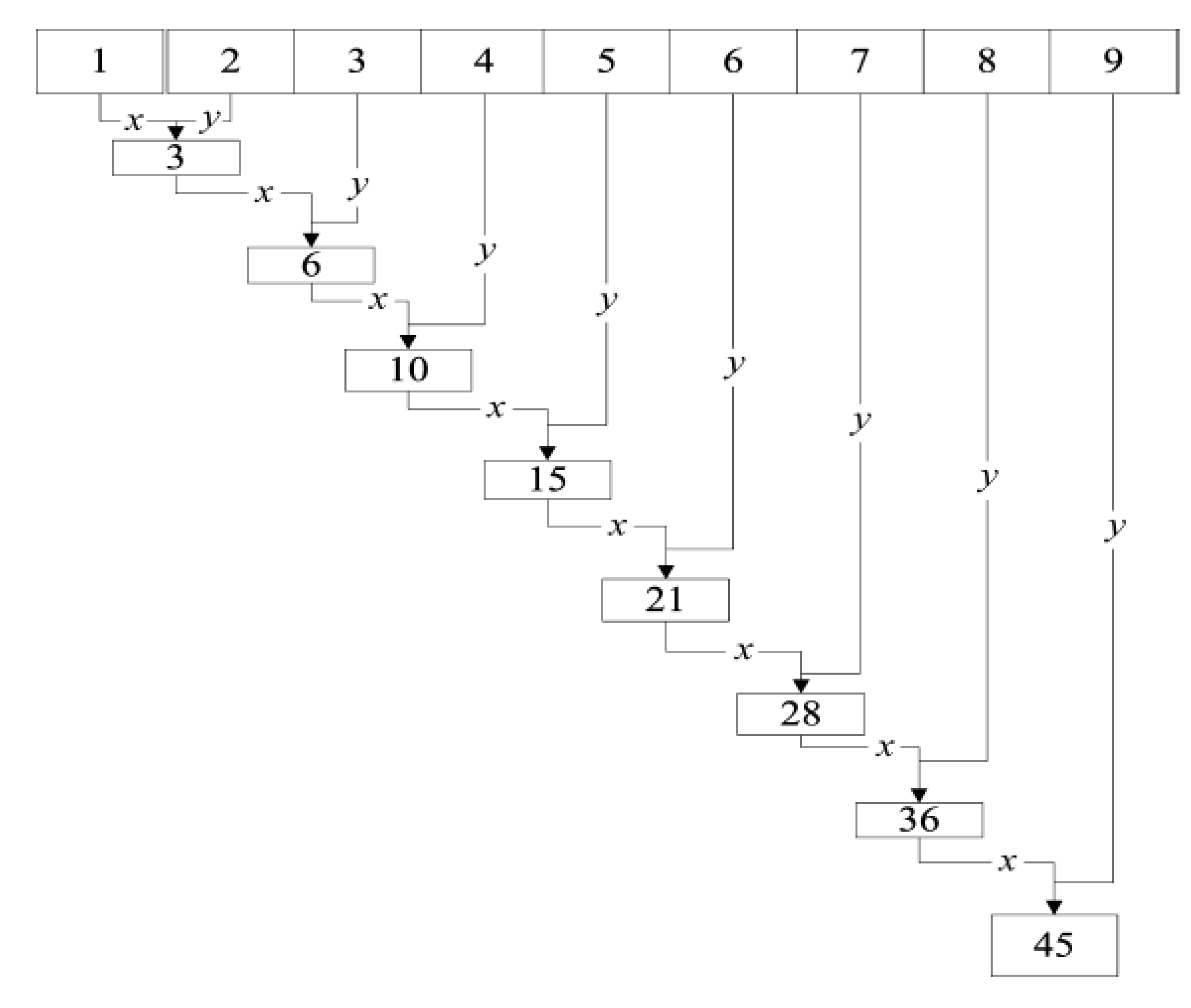


图 2-1　reduce( )函数执行过程示意图

与上面代码中最后演示的用法类似，作为一种技巧，reduce( )函数还支持下面的用法(感谢浙江省浦江中学方春林老师提供本例用法)：

```
>>>from random import randint
>>>lst =[randint(1,10) for i in range(50)]             #随机数列表
>>>def tjNum(dic,k):                                   #统计元素出现次数
    if k in dic:
      dic[k] +=1
    else:
      dic[k] =1
    return dic
>>>reduce(tjNum,lst,{})
{1: 6,2: 3,3: 6,4: 3,5: 4,6: 7,7: 5,8: 5,9: 6,10: 5}
```

内置函数 filter( )将一个单参数函数作用到一个序列上，返回该序列中使得该函数返回值为 True 的那些元素组成的 filter 对象，如果指定函数为 None，则返回序列中等价于 True 的元素。

```
>>>seq=['foo','x41','?! ','* * * ']
>>>def func(x):
    return x.isalnum()                                 #测试是否为字母或数字
>>>filter(func,seq)                                    #返回 filter 对象
<filter object at 0x000000000305D898 >
>>>list(filter(func,seq))                                  #把 filter 对象转换为列表
['foo','x41']
>>>seq                                                     #不对原列表做任何修改
```



```
['foo','x41','?! ','* * * ']
>>>[x for x in seq if x.isalnum()]                #使用列表推导式实现相同功能
['foo','x41']
>>>list(filter(lambda x: x.isalnum(),seq))        #使用 lambda 表达式实现相同功能
['foo','x41']
>>>list(filter(None,[1,2,3,0,0,4,0,5]))           #指定函数为 None
[1,2,3,4,5]
```

2.4.7 range()

range()是 Python 开发中非常常用的一个内置函数,语法格式为 range([start,] end [, step]),有 range(stop)、range(start, stop)和 range(start, stop, step)3 种用法。该函数返回具有惰性求值特点的 range 对象,其中包含左闭右开区间[start,end)内以 step 为步长的整数。参数 start 默认为 0,step 默认为 1。

```
>>>range(5)                                       #start 默认为 0,step 默认为 1
range(0,5)
>>>list(_)
[0,1,2,3,4]
>>>list(range(1,10,2))                            #指定起始值和步长
[1,3,5,7,9]
>>>list(range(9,0,-2))                            #步长为负数时,start 应比 end 大
[9,7,5,3,1]
```

在循环结构中经常使用 range()函数来控制循环次数,例如:

```
>>>for i in range(4):                             #循环 4 次
    print(3,end=' ')
3 3 3 3
```

当然,也可以使用 range()函数来控制数值范围,例如,下面的程序片段可以用来输出 200 以内能被 17 整除的最大正整数。

```
for i in range(200,0,-1):
    if i%17==0:
        print(i)
        break
```

2.4.8 zip()

zip()函数用来把多个可迭代对象中的元素压缩到一起,返回一个可迭代的 zip 对象,其中每个元素都是包含原来的多个可迭代对象对应位置上元素的元组,最终结果中包含的元素个数取决于所有参数序列或可迭代对象中最短的那个。可以这样理解这个函数,把多个序列或可迭代对象中的所有元素左对齐,然后像拉拉链一样往右拉,把所经过的每



个序列中相同位置上的元素都放到一个元组中，只要有一个序列中的所有元素都处理完了就不再拉拉链了，返回包含若干元组的 zip 对象。

```
>>>list(zip('abcd',[1,2,3]))                #压缩字符串和列表
[('a',1),('b',2),('c',3)]
>>>list(zip('abcd'))                        #对 1 个序列也可以压缩
[('a',),('b',),('c',),('d',)]
>>>list(zip('123','abc',',.! '))            #压缩 3 个序列
[('1','a',','),('2','b','.'),('3','c','! ')]
>>>for item in zip('abcd',range(3)):        #zip 对象是可迭代的
    print(item)
('a',0)
('b',1)
('c',2)
>>>x=zip('abcd','1234')
>>>list(x)
[('a','1'),('b','2'),('c','3'),('d','4')]
>>>list(x)                                  #zip 对象只能遍历一次
[]
```

2.4.9 eval()、exec()

内置函数 eval()用来计算字符串的值，在有些场合也可以用来实现类型转换的功能，这个用法在 2.4.1 节和 2.4.3 节出现过多次，这里不再重复。除此之外，eval()也可以对字节串进行求值，还可以执行内置函数 compile()编译生成的代码对象。

```
>>>eval(b'3+5')
8
>>>eval(compile('print(3+5)','temp.txt','exec'))
8
>>>eval('9')                                #把数字字符串转换为数字
9
>>>eval('09')                               #抛出异常，不允许以 0 开头的数字
SyntaxError: invalid token
>>>int('09')                                #这样转换是可以的
9
```

另外，由于 eval()并不对参数字符串进行安全性检查，如果精心构造一些语句的话可能会引发安全漏洞，应尽量使用标准库 ast 提供的安全求值函数 literal_eval()。

```
>>>eval("__import__('os').startfile(r'C:\Windows\\notepad.exe')")
                                            #打开记事本程序
>>>import ast
>>>ast.literal_eval("__import__('os').startfile(r'C:\Windows\\notepad.exe')")
                                            #无法执行，引发异常
```

```
ValueError: malformed node or string: <_ast.Call object at 0x00000000033E2C18>
```

内置函数 exec( )用来执行指定的 Python 源代码或者由 compile( )编译的代码对象。

```
>>>exec('x=3')                                    #执行语句 x=3
>>>x
3
>>>exec('help(sum)')                              #查看内置函数 sum()的帮助文档
>>>obj=compile('for i in range(5):print(i,end=" ")','temp.txt','exec')
>>>obj                                            #内置函数 compile()生成的代码对象
<code object <module>at 0x00000000033A29C0,file "temp.txt",line 1>
>>>exec(obj)                                      #使用 exec()执行代码对象
0 1 2 3 4
>>>eval(obj)                                      #也可以使用 eval()执行代码对象
0 1 2 3 4
>>>obj=compile('x=666','temp.txt','exec')
>>>eval(obj)
>>>x
666
>>>for i,v in enumerate(range(5,10)):             #动态创建变量名
    exec('element'+str(i)+'='+str(v))

>>>element0
5
>>>element1
6
```

第 3 章



玄之又玄，众妙之门：详解 Python 序列结构

Python 中常用的序列结构有列表、元组、字典、字符串、集合等（虽然有人并不主张把字典和集合看作序列，但这真的不重要），从是否有序这个角度可以分为有序序列和无序序列，从是否可变来看则可以分为可变序列和不可变序列两大类，如图 3-1 所示。另外，生成器对象和 range、map、enumerate、filter、zip 等对象的某些用法也类似于序列，尽管这些对象更大的特点是惰性求值。列表、元组、字符串等有序序列以及 range 对象均支持双向索引，第一个元素下标为 0，第二个元素下标为 1，以此类推；如果使用负数作为索引，则最后一个元素下标为 -1，倒数第二个元素下标为 -2，以此类推。可以使用负整数作为索引是 Python 有序序列的一大特色，熟练掌握和运用可以大幅度提高开发效率。

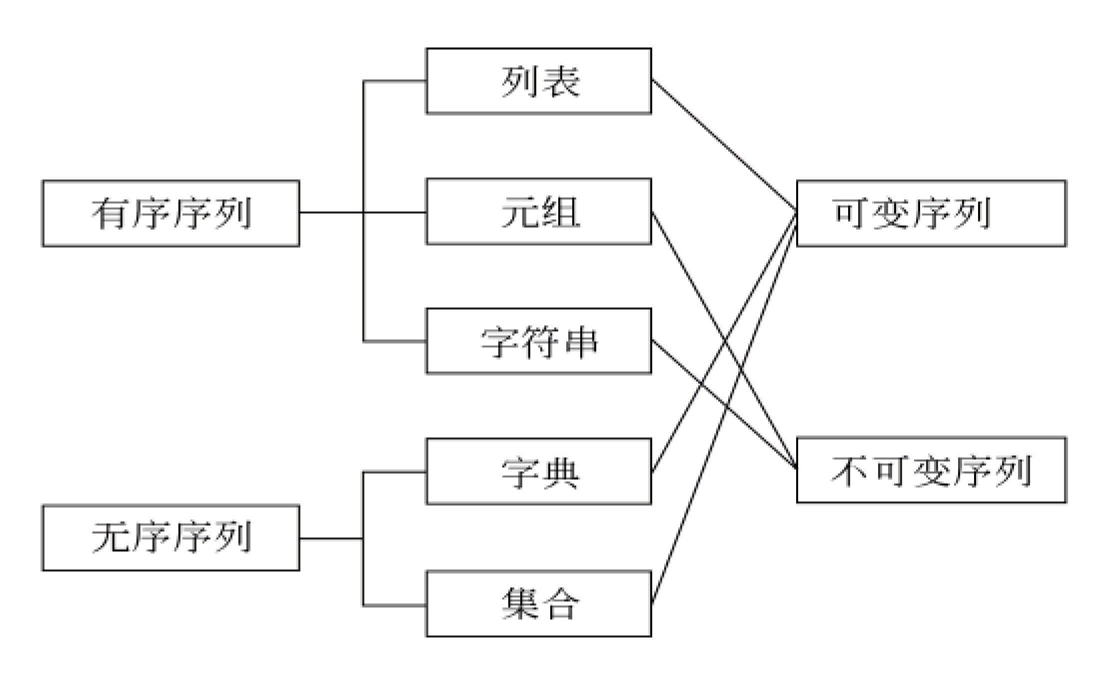


图 3-1　Python 序列分类示意图

3.1　列表：打了激素的数组

列表（list）是最重要的 Python 内置对象之一，是**包含若干元素的有序连续内存空间**。当列表增加或删除元素时，**列表对象自动进行内存的扩展或收缩**，从而**保证相邻元素之间没有缝隙**。Python 列表的这个内存自动管理功能可以大幅度减少程序员的负担，但插入和删除非尾部元素时涉及列表中大量元素的移动，会严重影响效率。另外，在非尾部位置插入和删除元素时会改变该位置后面的元素在列表中的索引，这对于某些操作可能会导致意外的错误结果。因此，除非确实有必要，否则应尽量从列表尾部进行元素的追加与删除操作。

在形式上，列表的所有元素放在一对方括号[ ]中，相邻元素之间使用逗号分隔。在Python中，同一个列表中元素的数据类型可以各不相同，可以同时包含整数、实数、字符串等基本类型的元素，也可以包含列表、元组、字典、集合、函数以及其他任意对象。如果只有一对方括号而没有任何元素则表示空列表。下面几个都是合法的列表对象：

```
[10,20,30,40]
['crunchy frog','ram bladder','lark vomit']
['spam',2.0,5,[10,20]]
[['file1',200,7],['file2',260,9]]
[{3},{5:6},(1,2,3)]
```

Python 采用基于值的自动内存管理模式，变量并不直接存储值，而是存储值的引用或内存地址，这也是 python 中变量可以随时改变类型的重要原因。同理，Python 列表中的元素也是值的引用，所以列表中各元素可以是不同类型的数据。

需要注意的是，列表的功能虽然非常强大，但是负担也比较重，开销较大，在实际开发中，最好根据实际的问题选择一种合适的数据类型，要尽量避免过多使用列表。

3.1.1 列表创建与删除

使用“=”直接将一个列表赋值给变量即可创建列表对象。

```
>>>a_list=['a','b','mpilgrim','z','example']
>>>a_list=[]                                   #创建空列表
```

也可以使用 list( )函数把元组、range 对象、字符串、字典、集合或其他可迭代对象转换为列表。需要注意的是，把字典转换为列表时默认是将字典的“键”转换为列表，而不是把字典的元素转换为列表，如果想把字典的元素转换为列表，需要使用字典对象的 items( )方法明确说明，当然也可以使用 values( )来明确说明要把字典的“值”转换为列表。

```
>>>list((3,5,7,9,11))                          #将元组转换为列表
[3,5,7,9,11]
>>>list(range(1,10,2))                         #将 range 对象转换为列表
[1,3,5,7,9]
>>>list('hello world')                         #将字符串转换为列表
['h','e','l','l','o',' ','w','o','r','l','d']
>>>list({3,7,5})                               #将集合转换为列表
[3,5,7]
>>>list({'a':3,'b':9,'c':78})                  #将字典的“键”转换为列表
['a','c','b']
>>>list({'a':3,'b':9,'c':78}.items())          #将字典的“键:值”对转换为列表
[('b',9),('c',78),('a',3)]
>>>x=list()                                    #创建空列表
```

当一个列表不再使用时，可以使用 del 命令将其删除，这一点适用于所有类型的

Python 对象。

```
>>>x=[1,2,3]
>>>del x                                    #删除列表对象
>>>x                                        #对象删除后无法再访问,抛出异常
NameError: name 'x' is not defined
```

严格来说，del 命令并不删除变量对应的值，只是删除变量并解除变量和值的绑定。Python 内部每个值都维护一个计数器，每当有新的变量引用该值时其引用计数器加 1，当该变量被删除或不再引用该值时其引用计数器减 1，当某个值的引用计数器变为 0 时则由垃圾回收器负责清理和删除。如果需要立刻进行垃圾回收，可以导入 gc 模块后调用其 collect( )方法。

```
>>>import sys
>>>sys.getrefcount(1)                       #查看值的引用次数
1243
>>>x=1
>>>sys.getrefcount(1)                       #有新变量引用该值,其引用计数器加 1
1244
>>>y=1
>>>sys.getrefcount(1)
1245
>>>del x                                    #删除变量并解除引用,该值的引用计数器减 1
>>>del y
>>>sys.getrefcount(1)
1243
>>>import gc
>>>gc.collect()                             #立刻进行垃圾回收,返回被清理的对象数量
0
```

3.1.2　列表元素访问

创建列表之后，可以**使用整数作为下标**来访问其中的元素，其中下标为 0 的元素表示第 1 个元素，下标为 1 的元素表示第 2 个元素，下标为 2 的元素表示第 3 个元素，以此类推；列表还支持使用负整数作为下标，其中下标为 –1 的元素表示最后一个元素，下标为 –2 的元素表示倒数第 2 个元素，下标为 –3 的元素表示倒数第 3 个元素，以此类推，如图 3-2 所示(以列表['P', 'y', 't', 'h', 'o', 'n']为例)。

```
>>>x=list('Python')                         #创建类别对象
>>>x
['P','y','t','h','o','n']
>>>x[0]                                     #下标为 0 的元素,第一个元素
'P'
>>>x[-1]                                    #下标为 -1 的元素,最后一个元素
'n'
```



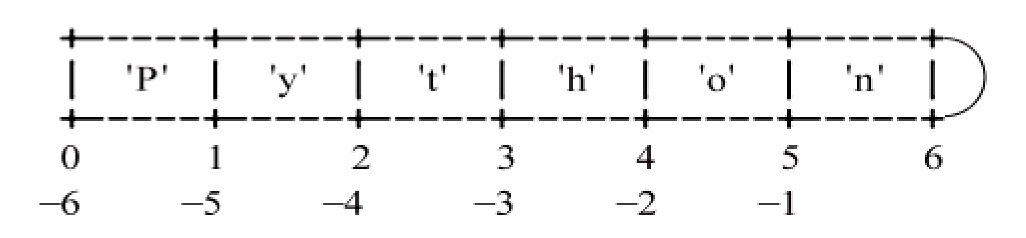


图 3-2 双向索引示意图

3.1.3 列表常用方法

列表、元组、字典、集合、字符串等 Python 序列有很多操作是通用的，而不同类型的序列又有一些特有的方法或者支持某些特有的运算符和内置函数。列表对象常用的方法如表 3-1 所示。

表 3-1 列表对象常用的方法

方　法	说　　明
append(x)	将 x 追加至列表尾部
extend(L)	将列表 L 中所有元素追加至列表尾部
insert(index, x)	在列表 index 位置处插入 x，该位置后面的所有元素后移并且在列表中的索引加 1，如果 index 为正数且大于列表长度则在列表尾部追加 x，如果 index 为负数且小于列表长度的相反数则在列表头部插入元素 x
remove(x)	在列表中删除第一个值为 x 的元素，该元素之后所有元素前移并且索引减 1，如果列表中不存在 x 则抛出异常
pop([index])	删除并返回列表中下标为 index 的元素，如果不指定 index 则默认为 -1，弹出最后一个元素；如果弹出中间位置的元素则后面的元素索引减 1；如果 index 不是[-L, L)区间上的整数则抛出异常，L 表示列表长度
clear()	清空列表，删除列表中的所有元素，保留列表对象
index(x)	返回列表中第一个值为 x 的元素的索引，若不存在值为 x 的元素则抛出异常
count(x)	返回 x 在列表中的出现次数
reverse()	对列表所有元素进行原地逆序，首尾交换
sort(key = None, reverse = False)	对列表中的元素进行原地排序，key 用来指定排序规则，reverse 为 False 表示升序，True 表示降序
copy()	返回列表的浅复制

1. append()、insert()、extend()

这 3 个方法都可以用于向列表对象中添加元素，其中 append()用于向列表尾部追加一个元素，insert()用于向列表任意指定位置插入一个元素，extend()用于将另一个列表中的所有元素追加至当前列表的尾部。这 3 个方法都属于原地操作，不影响列表对象在内存中的起始地址。对于长列表而言，使用 insert()方法在列表首部或中间位置插入元素时效率较低。如果确实需要在首部按序插入多个元素的话，可以先在尾部追加，然后再



使用 reverse()方法进行翻转，或者考虑使用标准库 collections 中的双端队列 deque 对象提供的 appendleft()方法。

```
>>>x=[1,2,3]
>>>id(x)                                    #查看对象的内存地址
50159368
>>>x.append(4)                              #在尾部追加元素
>>>x.insert(0,0)                            #在指定位置插入元素
>>>x.extend([5,6,7])                        #在尾部追加多个元素
>>>x
[0,1,2,3,4,5,6,7]
>>>id(x)                                    #列表在内存中的地址不变
50159368
```

2. pop()、remove()、clear()

这 3 个方法用于删除列表中的元素，其中 pop()用于**删除并返回**指定位置(默认是最后一个)上的元素，如果指定的位置不是合法的索引则抛出异常，对空列表调用 pop()方法也会抛出异常；remove()用于删除列表中第一个值与指定值相等的元素，如果列表中不存在该元素则抛出异常；clear()用于清空列表中的所有元素。这 3 个方法也属于原地操作，不影响列表对象的内存地址。另外，还可以使用 del 命令删除列表中指定位置的元素，同样也属于原地操作。

```
>>>x=[1,2,3,4,5,6,7]
>>>x.pop()                                  #弹出并返回尾部元素
7
>>>x.pop(0)                                 #弹出并返回指定位置的元素
1
>>>x.clear()                                #删除所有元素
>>>x
[]
>>>x=[1,2,1,1,2]
>>>x.remove(2)                              #删除首个值为 2 的元素
>>>del x[3]                                 #删除指定位置上的元素
>>>x
[1,1,1]
```

必须要再次强调的是，由于**列表具有内存自动收缩和扩张功能**，在列表中间位置插入或删除元素时，不仅效率较低，该位置后面所有元素在列表中的索引也会发生变化，必须牢牢记住这一点。

3. count()、index()

列表方法 count()用于返回列表中指定元素出现的次数；index()用于返回指定元素

在列表中**首次出现**的位置,如果该元素不在列表中则抛出异常。

```
>>>x=[1,2,2,3,3,3,4,4,4,4]
>>>x.count(3)                       #元素 3 在列表 x 中的出现次数
3
>>>x.count(5)                       #不存在,返回 0
0
>>>x.index(2)                       #元素 2 在列表 x 中首次出现的索引
1
>>>x.index(5)                       #列表 x 中没有 5,抛出异常
ValueError: 5 is not in list
```

通过前面的介绍我们已经知道,列表对象的很多方法在特殊情况下会抛出异常,而一旦出现异常,整个程序就会崩溃,这是我们不希望的。为避免引发异常而导致程序崩溃,一般来说有两种方法:①使用选择结构确保列表中存在指定元素再调用有关的方法;②使用异常处理结构。下面的代码使用异常处理结构保证用户输入的是三位数,然后使用关键字 in 来测试用户输入的数字是否在列表中,如果存在则输出其索引,否则提示不存在。

```
from random import sample

lst=sample(range(100,1000),100)

while True:
    x=input('请输入一个三位数: ')
    try:
        assert len(x)==3,'长度必须为 3'
        x=int(x)
        break
    except:
        pass

if x in lst:
    print('元素 {0}在列表中的索引为: {1}'.format(x,lst.index(x)))
else:
    print('列表中不存在该元素.')
```

4. sort()、reverse()

列表对象的 sort()方法用于按照指定的规则对所有元素进行排序,默认规则是所有元素从小到大升序排序;reverse()方法用于将列表所有元素逆序或翻转,也就是第一个元素和倒数第一个元素交换位置,第二个元素和倒数第二个元素交换位置,以此类推。

```
>>>x=list(range(11))                #包含 11 个整数的列表
>>>import random
```



```
>>>random.shuffle(x)                          #把列表 x 中的元素随机乱序
>>>x
[6,0,1,7,4,3,2,8,5,10,9]
>>>x.sort(key=lambda item:len(str(item)),reverse=True)
                                              #按转换成字符串以后的长度降序排列
>>>x
[10,6,0,1,7,4,3,2,8,5,9]
>>>x.sort(key=str)                            #按转换为字符串后的大小升序排序
>>>x
[0,1,10,2,3,4,5,6,7,8,9]
>>>x.sort()                                   #按默认规则排序
>>>x
[0,1,2,3,4,5,6,7,8,9,10]
>>>x.reverse()                                #把所有元素翻转或逆序
>>>x
[10,9,8,7,6,5,4,3,2,1,0]
```

列表对象的 sort( ) 和 reverse( ) 分别对列表进行原地排序(in-place sorting)和逆序，没有返回值。所谓“原地”，意思是用处理后的数据替换原来的数据，列表首地址不变，列表中元素原来的顺序全部丢失。

如果不想丢失原来的顺序，可以使用 2.4.4 节介绍的内置函数 sorted( ) 和 reversed( )。其中，内置函数 sorted( ) 返回排序后的新列表，参数 key 和 reverse 的含义与列表方法 sort( ) 完全相同；内置函数 reversed( ) 返回一个逆序后的 reversed 对象。充分利用列表对象的 sort( ) 方法和内置函数 sorted( ) 的 key 参数，可以实现更加复杂的排序，以内置函数 sorted( ) 为例：

```
>>>gameresult=[['Bob',95.0,'A'],
              ['Alan',86.0,'C'],
              ['Mandy',83.5,'A'],
              ['Rob',89.3,'E']]
>>>from operator import itemgetter
>>>sorted(gameresult,key=itemgetter(2))      #按子列表第 3 个元素进行升序排序
[['Bob',95.0,'A'],['Mandy',83.5,'A'],['Alan',86.0,'C'],['Rob',89.3,'E']]
>>>sorted(gameresult,key=itemgetter(2,0)) #先按第 3 个元素升序并排列，再按第一个元素升序排序
[['Bob',95.0,'A'],['Mandy',83.5,'A'],['Alan',86.0,'C'],['Rob',89.3,'E']]
>>>sorted(gameresult,key=itemgetter(2,0),reverse=True)
[['Rob',89.3,'E'],['Alan',86.0,'C'],['Mandy',83.5,'A'],['Bob',95.0,'A']]
>>>list1=["what","I'm","sorting","by"]        #以一个列表内容为依据
>>>list2=["something","else","to","sort"]     #对另一个列表内容进行排序
>>>pairs=zip(list1,list2)                     #把两个列表中的对应位置元素配对
>>>[item[1] for item in sorted(pairs,key=lambda x:x[0],reverse=True)]
['something','to','sort','else']
>>>x=[[1,2,3],[2,1,4],[2,2,1]]
>>>sorted(x,key=lambda item:(item[1],-item[2]))
```



```
                                                  #以第 2 个元素升序
                                                  #第 3 个元素降序排序
                                                  #这里的负号只适用于数值型元素
[[2,1,4],[1,2,3],[2,2,1]]
>>>x=['aaaa','bc','d','b','ba']
>>>sorted(x,key=lambda item: (len(item),item))
                                                  #先按长度排序,长度一样的正常排序
['b','d','ba','bc','aaaa']
```

5. copy()

列表对象的 copy()方法返回列表的浅复制。所谓浅复制,是指生成一个新的列表,并且把原列表中所有元素的引用都复制到新列表中。如果原列表中只包含整数、实数、复数等基本类型或元组、字符串这样的不可变类型的数据,一般是没有问题的。但是,如果原列表中包含列表之类的可变数据类型,由于浅复制时只是把子列表的引用复制到新列表中,于是修改任何一个都会影响另外一个。

```
>>>x=[1,2,[3,4]]                                  #原列表中包含子列表
>>>y=x.copy()                                     #浅复制
>>>y                                              #两个列表中的内容看起来完全一样
[1,2,[3,4]]
>>>y[2].append(5)                                 #为新列表中的子列表追加元素
>>>x[0]=6                                         #整数、实数等不可变类型不受此影响
>>>y.append(6)                                    #在新列表尾部追加元素
>>>y
[1,2,[3,4,5],6]
>>>x                                              #原列表不受影响
[6,2,[3,4,5]]
```

列表对象的 copy()方法和切片操作以及标准库 copy 中的 copy()函数一样都是返回浅复制,如果想避免上面代码演示的问题,可以使用标准库 copy 中的 deepcopy()函数实现深复制。所谓深复制,是指对原列表中的元素进行递归,把所有的值都复制到新列表中,对嵌套的子列表不再是复制引用。这样一来,新列表和原列表是互相独立,修改任何一个都不会影响另外一个。

```
>>>import copy
>>>x=[1,2,[3,4]]
>>>y=copy.deepcopy(x)                             #深复制
>>>x[2].append(5)                                 #为原列表中的子列表追加元素
>>>y.append(6)                                    #在新列表尾部追加元素
>>>y
[1,2,[3,4],6]
>>>x
[1,2,[3,4,5]]
```



不论是浅复制还是深复制，与列表对象的直接赋值都是不一样的情况。下面的代码把同一个列表赋值给两个不同的变量，这两个变量是互相独立的，修改任何一个都不会影响另外一个。

```
>>>x=[1,2,[3,4]]
>>>y=[1,2,[3,4]]                    #把同一个列表对象赋值给两个变量
>>>x.append(5)
>>>x[2].append(6)                   #修改其中一个列表的子列表
>>>x
[1,2,[3,4,6],5]
>>>y                                #不影响另外一个列表
[1,2,[3,4]]
```

下面的代码演示的是另外一种情况，把一个列表变量赋值给另外一个变量，这样两个变量指向同一个列表对象，对其中一个做的任何修改都会立刻在另外一个变量得到体现。

```
>>>x=[1,2,[3,4]]
>>>y=x                              #两个变量指向同一个列表
>>>x[2].append(5)
>>>x.append(6)
>>>x[0]=7
>>>x
[7,2,[3,4,5],6]
>>>y                                #对 x 做的任何修改,y 都会受到影响
[7,2,[3,4,5],6]
```

3.1.4　列表对象支持的运算符

加法运算符（+）也可以实现列表增加元素的目的，但这个运算符不属于原地操作，而是返回新列表，并且涉及大量元素的复制，效率非常低。使用复合赋值运算符 += 实现列表追加元素时属于原地操作，与 append() 方法一样高效。

```
>>>x=[1,2,3]
>>>id(x)
53868168
>>>x=x+[4]                          #连接两个列表
>>>x
[1,2,3,4]
>>>id(x)                            #内存地址发生改变
53875720
>>>x+=[5]                           #为列表追加元素
>>>x
[1,2,3,4,5]
>>>id(x)                            #内存地址不变
53875720
```



乘法运算符 * 可以用于列表和整数相乘,表示序列重复,返回新列表,从一定程度上来说也可以实现为列表增加元素的功能。与加法运算符(+)一样,该运算符也适用于元组和字符串。另外,运算符 *= 也可以用于列表元素重复,与运算符 += 一样属于原地操作。

```
>>>x=[1,2,3,4]
>>>id(x)
54497224
>>>x=x*2                                #元素重复,返回新列表
>>>x
[1,2,3,4,1,2,3,4]
>>>id(x)                                #地址发生改变
54603912
>>>x*=2                                 #元素重复,原地进行
>>>x
[1,2,3,4,1,2,3,4,1,2,3,4,1,2,3,4]
>>>id(x)                                #地址不变
54603912
>>>[1,2,3]*0                            #重复 0 次,清空
[]
```

由于 Python 列表中元素存储的是地址而不是值,当包含子列表的列表进行元素重复的时候,情况会复杂一些。

```
>>>x=[[1]]*3
>>>x
[[1],[1],[1]]
>>>id(x[0])==id(x[1])==id(x[2])                 #新列表 x 中的 3 个元素是同一个列表对象
True
>>>x[0].append(3)                       #为其中一个子列表追加新元素
>>>x                                    #另外两个子列表会受到同样的影响
[[1,3],[1,3],[1,3]]
>>>x[0]=[1,2,3]                         #直接修改第一个元素的值
>>>x                                    #不影响另外两个元素
[[1,2,3],[1,3],[1,3]]
>>>id(x[1])==id(x[2])
True
>>>id(x[0])==id(x[1])                   #不再是同一个对象
False
```

不过,上面的描述并不适用于下面的情况:

```
>>>x=[[] for i in range(3)]             #列表推导式
>>>x
[[],[],[]]
>>>x[0].append(1)                       #3 个子列表互不影响
>>>x[1].append(3)
```




```
>>>x[2].append(5)
>>>x
[[1],[3],[5]]
```

成员测试运算符 in 可用于测试列表中是否包含某个元素，查询时间随着列表长度的增加而线性增加，而同样的操作对于集合而言则是常数级的。

```
>>>3 in [1,2,3]
True
>>>3 in [1,2,'3']
False
```

3.1.5　内置函数对列表的操作

除了列表对象自身方法之外，很多 Python 内置函数也可以对列表进行操作。例如，max()、min()函数用于返回列表中所有元素的最大值和最小值，sum()函数用于返回列表中所有元素之和，len()函数用于返回列表中元素个数，zip()函数用于将多个列表中元素重新组合为元组并返回包含这些元组的 zip 对象，enumerate()函数返回包含若干下标和值的迭代对象，map()函数把函数映射到列表上的每个元素，filter()函数根据指定函数的返回值对列表元素进行过滤，all()函数用来测试列表中是否所有元素都等价于 True，any()用来测试列表中是否有等价于 True 的元素。另外，标准库 functools 中的 reduce()函数以及标准库 itertools 中的 compress()、groupby()、dropwhile()等大量函数也可以对列表进行操作。这里重点介绍内置函数对列表的操作，关于 reduce()函数请参考第 2 章的介绍，标准库 itertools 的用法请参考第 4 章。

```
>>>x=list(range(11))                  #生成列表
>>>import random
>>>random.shuffle(x)                  #打乱列表中元素的顺序
>>>x
[0,6,10,9,8,7,4,5,2,1,3]
>>>all(x)                             #测试是否所有元素都等价于 True
False
>>>any(x)                             #测试是否存在等价于 True 的元素
True
>>>max(x)                             #返回最大值
10
>>>max(x,key=str)                     #按指定规则返回最大值
9
>>>min(x)
0
>>>sum(x)                          #所有元素之和
55
>>>len(x)                          #列表元素个数
```



```
11
>>>list(zip(x, [1] * 11))                #多列表元素重新组合
[(0,1), (6,1), (10,1), (9,1), (8,1), (7,1), (4,1), (5,1), (2,1), (1,1), (3,1)]
>>>list(zip(range(1,4)))                 #zip()函数也可以用于一个序列或迭代对象
[(1,), (2,), (3,)]
>>>list(zip(['a','b','c'], [1,2]))       #如果两个列表不等长,以短的为准
[('a',1), ('b',2)]
>>>enumerate(x)                          #枚举列表元素,返回 enumerate 对象
<enumerate object at 0x00000000030A9120>
>>>list(enumerate(x))                    #enumerate 对象可以转换为列表、元组、集合
[(0,0), (1,6), (2,10), (3,9), (4,8), (5,7), (6,4), (7,5), (8,2), (9,1), (10,3)]
```

3.1.6 使用列表模拟向量运算

3.1.4 节已经介绍过,Python 列表支持与整数的乘法运算,表示列表元素进行重复并生成新列表。Python 列表不支持与整数的加、减、除运算,也不支持列表之间的减、乘、除操作。列表之间的加法运算表示列表元素的合并,生成新列表。

```
>>>[1,2,3] +[4,5,6]
[1,2,3,4,5,6]
```

然而,向量运算经常涉及这样的操作,例如向量所有分量同时加、减、乘、除同一个数,或者向量之间的加、减、乘运算,Python 列表对象本身不支持这样的操作,不过可以借助于内置函数、列表推导式和标准库 operator 中的方法来实现。如果需要更加丰富和强大的向量或矩阵运算,可以借助于 Python 扩展库 numpy 实现。

```
>>>from random import randint
>>>x =[randint(1,100) for i in range(10)] #生成 10 个[1,100]区间内的随机数
>>>x
[46,76,47,28,5,15,57,29,9,40]
>>>list(map(lambda i: i +5,x))           #所有元素同时加 5
[51,81,52,33,10,20,62,34,14,45]
>>>[i +5 for i in x]                     #使用列表推导式实现同样的功能
[51,81,52,33,10,20,62,34,14,45]
>>>x =[randint(1,10) for i in range(10)] #生成两个列表
>>>y =[randint(1,10) for i in range(10)]
>>>x
[2,2,9,6,7,9,2,1,2,7]
>>>y
[8,1,9,7,1,5,8,4,1,9]
>>>import operator
>>>sum(map(operator.mul,x,y))            #向量内积
278
>>>sum((i * j for i,j in zip(x,y)))      #使用内置函数计算向量内积
```



```
278
>>>list(map(operator.add,x,y))           #两个等长的向量对应元素相加
[10,3,18,13,8,14,10,5,3,16]
>>>list(map(lambda i,j: i+j,x,y))        #使用 lambda 表达式实现同样的功能
[10,3,18,13,8,14,10,5,3,16]
>>>[i+j for i,j in zip(x,y)]             #使用列表推导式实现同样的功能
[10,3,18,13,8,14,10,5,3,16]
```

3.1.7 列表推导式语法与应用案例

列表推导式(list comprehension),也称为列表解析式,可以使用非常简洁的方式对列表或其他可迭代对象的元素进行遍历、过滤或再次计算,快速生成满足特定需求的新列表,代码非常简洁,具有很强的可读性,是 Python 程序开发时应用最多的技术之一。Python 的内部实现对列表推导式做了大量优化,可以保证很快的运行速度,也是推荐使用的一种技术。列表推导式的语法形式为

```
[expression for expr1 in sequence1 if condition1
            for expr2 in sequence2 if condition2
            for expr3 in sequence3 if condition3
             ⋮
            for exprN in sequenceN if conditionN]
```

列表推导式在逻辑上等价于一个循环语句,只是形式上更加简洁。例如:

```
>>>aList=[x*x for x in range(10)]
```

相当于

```
>>>aList=[]
>>>for x in range(10):
    aList.append(x*x)
```

当然,如果不使用列表推导式的话,也可以借助于 Python 函数式编程的特点使用下面的代码实现同样的功能。

```
>>>aList=list(map(lambda x: x*x,range(10)))
>>>aList=list(map(lambda x: pow(x,2),range(10)))
```

再例如:

```
>>>freshfruit=[' banana',' loganberry ','passion fruit ']
>>>aList=[w.strip() for w in freshfruit]
```

等价于下面的代码:

```
>>>aList=[]
>>>for item in freshfruit:
    aList.append(item.strip())
```

当然也等价于：

```
>>>aList =list(map(lambda x: x.strip(),freshfruit))
```

或

```
>>>aList =list(map(str.strip,freshfruit))
```

大家应该听过一个故事，说是阿凡提（也有的说是阿基米德，这不是重点）与国王比赛下棋，国王说要是自己输了的话阿凡提想要什么他都可以拿得出来。阿凡提说那就要点米吧，棋盘一共 64 个小格子，在第一个格子里放 1 粒米，第二个格子里放 2 粒米，第三个格子里放 4 粒米，第四个格子里放 8 粒米，以此类推，后面每个格子里的米都是前一个格子里的 2 倍，一直把 64 个格子都放满。那么到底需要多少粒米呢？使用列表推导式再结合内置函数 sum()就很容易知道答案。

```
>>>sum([2 **i for i in range(64)])
18446744073709551615
```

按一斤大米约 26 000 粒计算，为放满棋盘，需要大概 350 亿吨大米。结果可想而知，最后国王没有办法拿出那么多米。

接下来再通过几个示例来进一步展示列表推导式的强大功能。

1. 实现嵌套列表的平铺

```
>>>vec =[[1,2,3],[4,5,6],[7,8,9]]
>>>[num for elem in vec for num in elem]
[1,2,3,4,5,6,7,8,9]
```

在这个列表推导式中有两个循环，其中第一个循环可以看作是外循环，执行得慢；第二个循环可以看作是内循环，执行得快。上面代码的执行过程等价于下面的写法：

```
>>>vec =[[1,2,3],[4,5,6],[7,8,9]]
>>>result =[]
>>>for elem in vec:
    for num in elem:
        result.append(num)
>>>result
[1,2,3,4,5,6,7,8,9]
```

如果不使用列表推导式的话，也可以借助于标准库 itertools 中的 chain()函数把子列表串起来成为一个列表。

```
>>>vec =[[1,2,3],[4,5,6],[7,8,9]]
>>>from itertools import chain
>>>list(chain(*vec))
[1,2,3,4,5,6,7,8,9]
```

当然，这里演示的只是一层嵌套列表的平铺，如果有多级嵌套或者不同子列表嵌套深



度不同的话，就不能使用上面的思路了。这时，可以使用函数递归实现。

```
def flatList(lst):
    result=[]                       #存放最终结果的列表
    def nested(lst):                #函数嵌套定义
        for item in lst:
            if isinstance(item,list):
                nested(item)        #递归子列表
            else:
                result.append(item) #扁平化列表
    nested(lst)                     #调用嵌套定义的函数
    return result                   #返回结果
```

另外，作为一个扩充，下面的代码可以看作上面嵌套列表平铺的一个逆运算，用来把一维列表转换成包含 r 个子列表的嵌套列表，每个子列表中包含 c 个元素，并且新列表恰好容纳原列表中的所有元素。如果需要更加复杂的数组和矩阵运算功能，可以借助于 Python 扩展库 numpy。

```
def resize(lst,r,c=-1):
    '''把一维列表转换成 r 行 c 列的嵌套列表'''
    #第一个参数必须是列表，并且只包含数字
    if not isinstance(lst,list):
        return 'must be a list'
    for item in lst:
        if not isinstance(item,(int,float,complex)):
            return 'must be a list of numbers'

    #第二个和第三个参数必须是整数
    if not (isinstance(r,int) and isinstance(c,int)):
        return 'Wrong size.'

    #原列表长度
    originLen=len(lst)

    #新的大小恰好能够容纳原列表中的所有元素
    if c==-1:
        if originLen%r !=0:
            return 'Wrong size.'
        c=originLen // r
    else:
        if r*c !=originLen:
            return 'Wrong size.'

    #使用切片生成新的嵌套列表
    result=[]
```

```
    for i in range(r):
        result.append(lst[i * c:i * c +c])

    #返回新列表
    return result

#测试
lst =list(range(20))
result =resize(lst,4,5)
if type(result) ==list:
    for row in result:
        print(row)
else:
    print(result)
```

2. 过滤不符合条件的元素

在列表推导式中可以使用 if 子句对列表中的元素进行筛选，只在结果列表中保留符合条件的元素。下面的代码可以列出当前文件夹下所有 Python 源文件：

```
>>>import os
>>>[filename for filename in os.listdir('.') if filename.endswith(('.py','.pyw'))]
```

下面的代码用于从列表中选择符合条件的元素组成新的列表：

```
>>>aList =[-1, -4,6,7.5, -2.3,9, -11]
>>>[i for i in aList if i >0]                    #所有大于 0 的数字
[6,7.5,9]
```

再例如，已知有一个包含一些同学成绩的字典，现在需要计算所有成绩的最高分、最低分、平均分，并查找所有最高分同学，代码可以这样编写：

```
>>>scores ={"Zhang San": 45,"Li Si": 78,"Wang Wu": 40,"Zhou Liu": 96,
            "Zhao Qi": 65,"Sun Ba": 90,"Zheng Jiu": 78,"Wu Shi": 99,
            "Dong Shiyi": 60}
>>>highest =max(scores.values())                 #最高分
>>>lowest =min(scores.values())                  #最低分
>>>average =sum(scores.values())/len(scores)     #平均分
>>>highest,lowest,average
(99,40,72.33333333333333)
>>>highestPerson =[name for name,score in scores.items() if score ==highest]
>>>highestPerson
['Wu Shi']
```

与上面的代码功能类似，下面的代码使用列表推导式查找列表中最大元素的所有位置：

```
>>>from random import randint
```




```
>>>x=[randint(1,10) for i in range(20)]                     #20 个介于 [1,10]的整数
>>>x
[10,2,3,4,5,10,10,9,2,4,10,8,2,2,9,7,6,2,5,6]
>>>m=max(x)
>>>[index for index,value in enumerate(x) if value==m]             #最大整数的所有出现
位置
[0,5,6,10]
```

3. 同时遍历多个列表或可迭代对象

```
>>>[(x,y) for x in [1,2,3] for y in [3,1,4] if x !=y]
[(1,3),(1,4),(2,3),(2,1),(2,4),(3,1),(3,4)]
>>>[(x,y) for x in [1,2,3] if x==1 for y in [3,1,4] if y!=x]
[(1,3),(1,4)]
```

对于包含多个循环的列表推导式，一定要清楚多个循环的执行顺序或“嵌套关系”。例如，上面第一个列表推导式等价于

```
>>>result=[]
>>>for x in [1,2,3]:
    for y in [3,1,4]:
        if x !=y:
            result.append((x,y))
>>>result
[(1,3),(1,4),(2,3),(2,1),(2,4),(3,1),(3,4)]
```

分析上面的代码和运行结果可以看出，这是两个序列元素笛卡儿积的一部分，作为一种技巧，也可以使用下面的代码实现同样的功能。

```
>>>import itertools
>>>list(itertools.product([1,2,3],[3,1,4]))
[(1,3),(1,1),(1,4),(2,3),(2,1),(2,4),(3,3),(3,1),(3,4)]
>>>list(filter(lambda x:x[0]!=x[1],itertools.product([1,2,3],[3,1,4])))
[(1,3),(1,4),(2,3),(2,1),(2,4),(3,1),(3,4)]
```

4. 使用列表推导式实现矩阵转置

```
>>>matrix=[[1,2,3,4],[5,6,7,8],[9,10,11,12]]
>>>[[row[i] for row in matrix] for i in range(4)]
[[1,5,9],[2,6,10],[3,7,11],[4,8,12]]
```

对于嵌套了列表推导式的列表推导式，一定要清楚其执行顺序。例如，上面列表推导式的执行过程等价于下面的代码

```
>>>matrix=[[1,2,3,4],[5,6,7,8],[9,10,11,12]]
>>>result=[]
>>>for i in range(len(matrix[0])):
```

```
    result.append([row[i] for row in matrix])
>>>result
[[1,5,9],[2,6,10],[3,7,11],[4,8,12]]
```

如果把内层的列表推导式也展开的话,完整的执行过程可以通过下面的代码来模拟:

```
>>>matrix=[ [1,2,3,4],[5,6,7,8],[9,10,11,12]]
>>>result=[]
>>>for i in range(len(matrix[0])):
  temp=[]
  for row in matrix:
    temp.append(row[i])
  result.append(temp)
>>>result
[[1,5,9],[2,6,10],[3,7,11],[4,8,12]]
```

当然,也可以使用内置函数 zip( )和 list( )来实现矩阵转置:

```
>>>list(map(list,zip(*matrix)))
[[1,5,9],[2,6,10],[3,7,11],[4,8,12]]
```

5. 列表推导式中可以使用函数或复杂表达式

```
>>>def f(v):
    if v%2==0:
        v=v**2
    else:
        v=v+1
    return v
>>>print([f(v) for v in [2,3,4,-1] if v>0])
[4,4,16]
>>>print([v**2 if v%2==0 else v+1 for v in [2,3,4,-1] if v>0])
[4,4,16]
```

6. 列表推导式支持文件对象迭代

```
>>>with open('C:\\RHDSetup.log','r') as fp:                #为节约篇幅,略去输出
   结果
   print([line for line in fp])
```

7. 使用列表推导式生成 100 以内的所有素数

```
>>>[p for p in range(2,100) if 0 not in [p%d for d in range(2,int(sqrt(p))+1)]]
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97]
```



另外，从 Python 3.6.x 开始支持在协程函数中使用下面形式的异步列表推导式，同样的用法也适用于字典推导式和集合推导式以及生成器推导式。关于协程函数的介绍请参考本书 5.8 节和 12.3 节的内容。

```
async def ticker(delay,to)                                  #协程函数,异步生成器
    for i in range(to):
        yield i
        await asyncio.sleep(delay)

async def run():
    result=[i async for i in ticker(1,10) if i%2]          #异步列表推导式
    print(result)

import asyncio
loop=asyncio.get_event_loop()
try:
    loop.run_until_complete(run())
finally:
    loop.close()
```

最后，Python 3.6.x 及更新版本还支持下面的用法，在列表推导式和其他类型推导式中使用 await 表达式。

```
import asyncio

async def half(x):
    return x/2

async def square(x):
    return x**2

async def cube(x):
    return x**3

async def run():
    result=[await f(3) for f in [half,square,cube]]
    print(result)

loop=asyncio.get_event_loop()
try:
    loop.run_until_complete(run())
finally:
    loop.close()
```

3.1.8 切片操作的强大功能

切片是 Python 序列的重要操作之一，除了适用于列表之外，还适用于元组、字符串、range 对象，但列表的切片操作具有最强大的功能。不仅可以使用切片来截取列表中的任何部分返回得到一个新列表，也可以通过切片来修改和删除列表中部分元素，甚至可以通过切片操作为列表对象增加元素。

在形式上，切片使用 2 个冒号分隔的 3 个数字来完成。

```
[start:end:step]
```

其中，3 个数字的含义与内置函数 range(start, end, step)完全一致，第一个数字 start 表示切片开始的位置，默认为 0；第二个数字 end 表示切片截止(但不包含)的位置(默认为列表长度)；第三个数字 step 表示切片的步长(默认为 1)。当 start 为 0 时可以省略，当 end 为列表长度时可以省略，当 step 为 1 时可以省略，省略步长时还可以同时省略最后一个冒号。另外，当 step 为负整数时，表示反向切片，这时 start 应该在 end 的右侧才行。

1. 使用切片获取列表部分元素

使用切片可以返回列表中部分元素组成的新列表。与使用索引作为下标访问列表元素的方法不同，切片操作不会因为下标越界而抛出异常，而是简单地在列表尾部截断或者返回一个空列表，代码具有更强的健壮性。

```
>>>aList =[3,4,5,6,7,9,11,13,15,17]
>>>aList[::]                           #返回包含原列表中所有元素的新列表
[3,4,5,6,7,9,11,13,15,17]
>>>aList[:: -1]                        #返回包含原列表中所有元素的逆序列表
[17,15,13,11,9,7,6,5,4,3]
>>>aList[::2]                          #隔一个取一个,获取偶数位置的元素
[3,5,7,11,15]
>>>aList[1::2]                         #隔一个取一个,获取奇数位置的元素
[4,6,9,13,17]
>>>aList[3:6]                          #指定切片的开始和结束位置
[6,7,9]
>>>aList[0:100]                        #切片结束位置大于列表长度时,从列表尾部截断
[3,4,5,6,7,9,11,13,15,17]
>>>aList[100]                          #抛出异常,不允许越界访问
IndexError: list index out of range
>>>aList[100:]                         #切片开始位置大于列表长度时,返回空列表
[]
>>>aList[ -15:3]                       #进行必要的截断处理
[3,4,5]
>>>len(aList)
10
```

```
>>>aList[3: -10: -1]                   #位置 3 在位置 -10 的右侧，-1 表示反向切片
[6,5,4]
>>>aList[3: -5]                        #位置 3 在位置 -5 的左侧，正向切片
[6,7]
```

2. 使用切片为列表增加元素

可以使用切片操作在列表任意位置插入新元素，不影响列表对象的内存地址，属于原地操作。

```
>>>aList =[3,5,7]
>>>aList[len(aList):]
[]
>>>aList[len(aList):] =[9]             #在列表尾部增加元素
>>>aList[:0] =[1,2]                    #在列表头部插入多个元素
>>>aList[3:3] =[4]                     #在列表中间位置插入元素
>>>aList
[1,2,3,4,5,7,9]
```

3. 使用切片替换和修改列表中的元素

```
>>>aList =[3,5,7,9]
>>>aList[:3] =[1,2,3]                  #替换列表元素，等号两边的列表长度相等
>>>aList
[1,2,3,9]
>>>aList[3:] =[4,5,6]                  #切片连续，等号两边的列表长度可以不相等
>>>aList
[1,2,3,4,5,6]
>>>aList[::2] =[0] * 3                 #隔一个修改一个
>>>aList
[0,2,0,4,0,6]
>>>aList[::2] =['a','b','c']           #隔一个修改一个
>>>aList
['a',2,'b',4,'c',6]
>>>aList[1::2] =range(3)               #序列解包的用法
>>>aList
['a',0,'b',1,'c',2]
>>>aList[1::2] =map(lambda x: x! =5,range(3))
>>>aList
['a',True,'b',True,'c',True]
>>>aList[1::2] =zip('abc',range(3))    #map、filter、zip 对象都支持这样的用法
>>>aList
['a',('a',0),'b',('b',1),'c',('c',2)]
>>>aList[::2] =[1]                     #切片不连续时等号两边列表的长度必须相等
ValueError: attempt to assign sequence of size 1 to extended slice of size 3
```



4. 使用切片删除列表中的元素

```
>>>aList=[3,5,7,9]
>>>aList[:3]=[]                    #删除列表中前 3 个元素
>>>aList
[9]
```

另外,也可以结合使用 del 命令与切片结合来删除列表中的部分元素,并且切片元素可以不连续。

```
>>>aList=[3,5,7,9,11]
>>>del aList[:3]                   #切片元素连续
>>>aList
[9,11]
>>>aList=[3,5,7,9,11]
>>>del aList[::2]                  #切片元素不连续,隔一个删一个
>>>aList
[5,9]
```

5. 切片得到的是列表的浅复制

在 3.1.3 节介绍列表对象的 copy()方法时曾经提到,切片返回的是列表元素的浅复制,与列表对象的直接赋值并不一样,和 3.1.3 节介绍的深复制也有本质的不同。

```
>>>aList=[3,5,7]
>>>bList=aList[::]                 #切片,浅复制
>>>aList==bList                    #两个列表的值相等
True
>>>aList is bList                  #浅复制,不是同一个对象
False
>>>id(aList)==id(bList)            #两个列表对象的地址不相等
False
>>>id(x[0])==id(y[0])              #相同的值在内存中只有一份
True
>>>bList[1]=8                      #修改 bList 列表元素的值不会影响 aList
>>>bList                           #bList 的值发生改变
[3,8,7]
>>>aList                           #aList 的值没有发生改变
[3,5,7]
>>>x=[[1],[2],[3]]                 #如果列表中包含列表或其他可变序列
>>>y=x[:]                          #情况会复杂一些
>>>y
[[1],[2],[3]]
>>>y[0]=[4]                        #直接修改 y 中下标为 0 的元素值,不影响 x
>>>y
```



```
[[4],[2],[3]]
>>>y[1].append(5)                #通过列表对象的方法原地增加元素
>>>y
[[4],[2,5],[3]]
>>>x                             #列表 x 也受到同样的影响
[[1],[2,5],[3]]
```

3.2　元组：轻量级列表

3.2.1　元组创建与元素访问

列表的功能虽然很强大，但负担也很重，在很大程度上影响了运行效率。有时候我们并不需要那么多功能，很希望能有个轻量级的列表，元组（tuple）正是这样一种类型。在形式上，元组的所有元素放在一对圆括号中，元素之间使用逗号分隔，如果元组中只有一个元素则必须在最后增加一个逗号。

```
>>>x=(1,2,3)                     #直接把元组赋值给一个变量
>>>type(x)                       #使用 type()函数查看变量类型
<class 'tuple'>
>>>x[0]                          #元组支持使用下标访问特定位置的元素
1
>>>x[-1]                         #最后一个元素，元组支持双向索引
3
>>>x[1]=4                        #元组是不可变的
TypeError: 'tuple' object does not support item assignment
>>>x=(3)                         #这和 x=3 是一样的
>>>x
3
>>>x=(3,)                        #如果元组中只有一个元素，必须在后面多写一个逗号
>>>x
(3,)
>>>x=()                          #空元组
>>>x=tuple()                     #空元组
>>>tuple(range(5))               #将其他迭代对象转换为元组
(0,1,2,3,4)
```

除了上面的方法可以直接创建元组之外，很多内置函数的返回值也是包含了若干元组的可迭代对象，例如 enumerate()、zip()等。

```
>>>list(enumerate(range(5)))
[(0,0),(1,1),(2,2),(3,3),(4,4)]
>>>list(zip(range(3),'abcdefg'))
[(0,'a'),(1,'b'),(2,'c')]
```

3.2.2 元组与列表的异同点

列表和元组都属于有序序列，都支持使用双向索引访问其中的元素，以及使用count( )方法统计元素的出现次数和index( )方法获取元素的索引，len( )、map( )、filter( )等大量内置函数和 +、*、+=、in 等运算符也都可以作用于列表和元组。虽然有着一定的相似之处，但列表和元组在本质上和内部实现上都有着很大的不同。

元组属于不可变（immutable）序列，不可以直接修改元组中元素的值，也无法为元组增加或删除元素。因此，元组没有提供 append( )、extend( )和 insert( )等方法，无法向元组中添加元素；同样，元组也没有 remove( )和 pop( )方法，也不支持对元组元素进行 del 操作，不能从元组中删除元素，而只能使用 del 命令删除整个元组。元组也支持切片操作，但是只能通过切片来访问元组中的元素，而不允许使用切片来修改元组中元素的值，也不支持使用切片操作来为元组增加或删除元素。从一定程度上讲，可以认为元组是轻量级的列表，或者"常量列表"。

Python 的内部实现对元组做了大量优化，访问速度比列表更快。如果定义了一系列常量值，主要用途仅是对它们进行遍历或其他类似用途，而不需要对其元素进行任何修改，那么一般建议使用元组而不用列表。元组在内部实现上不允许修改其元素值，从而使得代码更加安全，例如，调用函数时使用元组传递参数可以防止在函数中修改元组，而使用列表则很难保证这一点。

然而，虽然元组属于不可变序列，其元素的值是不可改变的，但是如果元组中包含可变序列，情况就复杂了。

```
>>>x=([1,2],3)                        #包含列表的元组
>>>x[0][0]=5                          #修改元组中的列表元素
>>>x[0].append(8)                     #为元组中的列表增加元素
>>>x
([5,2,8],3)
>>>x[0]=x[0]+[10]                     #试图修改元组的值，失败
TypeError: 'tuple' object does not support item assignment
>>>x
([5,2,8],3)
>>>x[0]+=[10]                         #抛出异常，但元组中的元素已被修改
TypeError: 'tuple' object does not support item assignment
>>>x
([5,2,8,10],3)
>>>y=x[0]                             #y 和 x[0]指向同一个列表
>>>y+=[11]                            #通过 y 可以影响元组 x 中的第一个列表
>>>x
([5,2,8,10,11],3)
>>>y=y+[12]                           #注意，这和 y+=[12]是有本质区别的
>>>y
[5,2,8,10,11,12]
```



```
>>>x
([5,2,8,10,11],3)
```

最后，作为不可变序列，与整数、字符串一样，元组可用作字典的键，也可以作为集合的元素。而列表则永远都不能当作字典键使用，也不能作为集合中的元素，因为列表不是不可变的。内置函数 hash( ) 可以用来测试一个对象是否可哈希。一般来说，并不需要关心该函数的返回值具体是什么，重点是对象是否可哈希，如果对象不可哈希会抛出异常。

```
>>>hash((1,))                          #元组、数字、字符串都是可哈希的
3430019387558
>>>hash(3)
3
>>>hash('hello world.')
-4012655148192931880
>>>hash([1,2])                         #列表不可哈希
TypeError: unhashable type: 'list'
```

3.2.3 生成器推导式

生成器推导式也称为生成器表达式(generator expression)，用法与列表推导式非常相似，在形式上生成器推导式使用圆括号(parentheses)作为定界符，而不是列表推导式所使用的方括号(square brackets)。与列表推导式最大的不同是，生成器推导式的结果是一个生成器对象。生成器对象类似于迭代器对象，具有惰性求值的特点，只在需要时生成新元素，比列表推导式具有更高的效率，空间占用非常少，尤其适合大数据处理的场合。

使用生成器对象的元素时，可以将其转化为列表或元组，也可以使用生成器对象的 __next__( ) 方法或者内置函数 next( ) 进行遍历，或者直接使用 for 循环来遍历其中的元素。但是不管用哪种形式，只能从前往后正向访问其中的元素，**没有任何方法可以再次访问已访问过的元素**，也不支持使用下标访问其中的元素。当所有元素访问结束以后，如果需要重新访问其中的元素，必须重新创建该生成器对象。enumerate、filter、map、zip 等对象也具有同样的特点。最后，包含 yield 语句的函数也可以用来创建生成器对象，详见第 5 章。

```
>>>g=((i+2)**2 for i in range(10))     #创建生成器对象
>>>g
<generator object <genexpr> at 0x0000000003095200>
>>>tuple(g)                            #将生成器对象转换为元组
(4,9,16,25,36,49,64,81,100,121)
>>>list(g)                             #生成器对象已遍历结束，没有元素了
[]
>>>g=((i+2)**2 for i in range(10))     #重新创建生成器对象
>>>g.__next__()                        #使用生成器对象的__next__()方法获取元素
4
>>>g.__next__()                        #获取下一个元素
```

```
9
>>>next(g)                              #使用函数 next()获取生成器对象中的元素
16
>>>g=((i+2)**2 for i in range(10))
>>>for item in g:                       #使用循环直接遍历生成器对象中的元素
  print(item,end=' ')
4 9 16 25 36 49 64 81 100 121
>>>x=filter(None,range(20))             #filter 对象也具有类似的特点
>>>1 in x
True
>>>5 in x
True
>>>2 in x                               #不可再次访问已访问过的元素
False
>>>x=map(str,range(20))                 #map 对象也具有类似的特点
>>>'0' in x
True
>>>'0' in x                             #不可再次访问已访问过的元素
False
```

与列表推导式不同，当生成器推导式中包含多个 for 语句时，在创建生成器对象时只对第一个 for 语句进行检查和计算，在调用内置函数 next( )或生成器对象的__next__( )方法获取值的时候才会检查和计算其他 for 语句。

```
>>>[x* y for x in range(3) for z in range(5)]
NameError: name 'y' is not defined
>>>g=(x* y for x in range(3) for z in range(5))
>>>next(g)                              #第二个 for 语句有问题，抛出异常
NameError: name 'y' is not defined
```

如果生成器推导式作为单参数函数时，可以省略两侧的圆括号。

```
>>>sum(x for x in range(3))
3
```

3.3 字典：反映对应关系的映射类型

字典（dict）是包含若干“键:值”元素的**无序可变**序列，字典中的每个元素包含用冒号分隔开的“键”和“值”两部分，表示一种映射或对应关系，也称为关联数组。定义字典时，每个元素的“键”和“值”之间用冒号分隔，不同元素之间用逗号分隔，所有的元素放在一对大括号“{}”中。

字典中元素的“键”可以是 Python 中任意不可变数据，例如整数、实数、复数、字符串、元组等类型等可哈希数据，但不能使用列表、集合、字典或其他可变类型作为字典的“键”。另外，字典中的“键”不允许重复，“值”是可以重复的。字典在内部维护的哈希表



使得检索操作非常快。使用内置字典类型 dict 时不要太在乎元素的先后顺序，如果确实在乎元素顺序可以使用 collections 的 OrderedDict 类。值得一提的是，在 Python 3.6 中又对内置类型 dict 进行了优化，比 Python 3.5.x 大概能节约 20% ~25% 的内存空间。

3.3.1　字典创建与删除

使用赋值运算符"="将一个字典赋值给一个变量即可创建一个字典变量。

```
>>>aDict={'server': 'db.diveintopython3.org','database': 'mysql'}
```

也可以使用内置类 dict 以不同形式创建字典，在第 2 章曾经介绍过这种用法，实际上是调用了 dict 类的构造方法。

```
>>>x=dict()                                       #空字典
>>>x={}                                           #空字典
>>>keys=['a','b','c','d']
>>>values=[1,2,3,4]
>>>dictionary=dict(zip(keys,values))              #根据已有数据创建字典
>>>d=dict(name='Dong',age=39)                     #以关键参数的形式创建字典
>>>aDict=dict.fromkeys(['name','age','sex'])
                                                  #以给定内容为"键"
                                                  #创建"值"为空的字典
>>>aDict
{'name': None,'age': None,'sex': None}
```

另外，Python 还支持使用字典推导式快速生成符合特定条件的字典。

```
>>>{i:str(i) for i in range(1,5)}
{1: '1',2: '2',3: '3',4: '4'}
>>>x=['A','B','C','D']
>>>y=['a','b','b','d']
>>>{i:j for i,j in zip(x,y)}
{'A': 'a','C': 'b','B': 'b','D': 'd'}
```

与其他类型的对象一样，当不再需要时，可以直接删除字典，不再赘述。

3.3.2　字典元素的访问

字典中的每个元素表示一种映射关系或对应关系，根据提供的"键"作为下标可以访问对应的"值"，如果字典中不存在这个"键"会抛出异常。

```
>>>aDict={'age': 39,'score': [98,97],'name': 'Dong','sex': 'male'}
>>>aDict['age']                                   #指定的"键"存在，返回对应的"值"
39
>>>aDict['address']                               #指定的"键"不存在，抛出异常
KeyError: 'address'
```

为了避免程序运行时引发异常而导致崩溃，在使用下标的方式访问字典元素时，最好

配合条件判断或者异常处理结构。

```
>>>aDict ={'age': 39,'score': [98,97],'name': 'Dong','sex': 'male'}
>>>if 'Age' in aDict:                          #首先判断字典中是否存在指定的"键"
    print(aDict['Age'])
else:
    print('Not Exists.')

Not Exists.
>>>try:                                        #使用异常处理结构
    print(aDict['address'])
except:
      print('Not Exists.')

Not Exists.
```

字典对象提供了一个 get( )方法用来返回指定"键"对应的"值",并且允许指定该键不存在时返回特定的"值"。例如:

```
>>>aDict.get('age')                            #如果字典中存在该"键"则返回对应的"值"
39
>>>aDict.get('address','Not Exists.')          #指定的"键"不存在时返回指定的默认值
'Not Exists.'
>>>import string
>>>import random
>>>x =string.ascii_letters +string.digits
>>>z =''.join((random.choice(x) for i in range(1000))) #生成 1000 个随机字符
>>>d=dict()
>>>for ch in z:                                #遍历字符串,统计频次
    d[ch] =d.get(ch,0) +1
>>>for k,v in sorted(d.items()):               #查看统计结果
    print(k,':',v)
```

字典对象的 setdefault( )方法用于返回指定"键"对应的"值",如果字典中不存在该"键",就添加一个新元素并设置该"键"对应的"值"(默认为 None)。

```
>>>aDict.setdefault('address','SDIBT')         #增加新元素
'SDIBT'
>>>aDict
{'age': 39,'score': [98,97],'name': 'Dong','address': 'SDIBT','sex': 'male'}
```

对字典对象直接进行迭代或者遍历时默认是遍历字典的"键",如果需要遍历字典的元素必须使用字典对象的 items( )方法明确说明,如果需要遍历字典的"值"则必须使用字典对象的 values( )方法明确说明。当使用 len( )、max( )、min( )、sum( )、sorted( )、enumerate( )、map( )、filter( )等内置函数以及成员测试运算符 in 对字典对象进行操作时,也遵循同样的约定。

```
>>>aDict={'age': 39,'score': [98,97],'name': 'Dong','sex': 'male'}
>>>for item in aDict:                          #默认遍历字典的“键”
    print(item,end=' ')
age score name sex
>>>for item in aDict.items():                  #明确指定遍历字典的元素
    print(item,end=' ')
('age',39) ('score',[98,97]) ('name','Dong') ('sex','male')
>>>aDict.items()
dict_items([('age',37),('score',[98,97]),('name','Dong'),('sex','male')])
>>>aDict.keys()
dict_keys(['age','score','name','sex'])
>>>aDict.values()
dict_values([37,[98,97],'Dong','male'])
```

3.3.3　元素的添加、修改与删除

当以指定“键”为下标为字典元素赋值时，有两种含义：①若该“键”存在，则表示修改该“键”对应的值；②若不存在，则表示添加一个新的“键:值”对，也就是添加一个新元素。

```
>>>aDict={'age': 35,'name': 'Dong','sex': 'male'}
>>>aDict['age']=39                             #修改元素值
>>>aDict['address']='SDIBT'                    #添加新元素
>>>aDict                                       #使用字典时并不需要太在意元素顺序
{'age': 39,'address': 'SDIBT','name': 'Dong','sex': 'male'}
```

使用字典对象的 update( ) 方法可以将另一个字典的“键:值”一次性全部添加到当前字典对象，如果两个字典中存在相同的“键”，则以另一个字典中的“值”为准对当前字典进行更新。

```
>>>aDict={'age': 37,'score': [98,97],'name': 'Dong','sex': 'male'}
>>>aDict.update({'a':97,'age':39})             #修改'age'键的值，同时添加新元素'a':97
>>>aDict
{'score': [98,97],'sex': 'male','a': 97,'age': 39,'name': 'Dong'}
```

字典对象的 setdefault( ) 方法也可以用来为字典添加新元素，3.3.2 节中已经介绍了该方法的用法。如果需要删除字典中指定的元素，可以使用 del 命令。

```
>>>del aDict['age']                            #删除字典元素
>>>aDict
{'score': [98,97],'sex': 'male','a': 97,'name': 'Dong'}
```

字典对象的 pop( ) 和 popitem( ) 方法可以弹出并删除指定的元素。

```
>>>aDict={'age': 37,'score': [98,97],'name': 'Dong','sex': 'male'}
>>>aDict.popitem()                             #弹出一个元素，对空字典会抛出异常
```

```
('age',37)                                 #在 Python 3.6.x 中结果略有不同,是正常的
>>>aDict.pop('sex')                        #弹出指定键对应的元素
'male'
>>>aDict
{'score': [98,97],'name': 'Dong'}
```

字典对象的 clear()方法用于清空字典对象中的所有元素;copy()方法返回字典对象的浅复制,关于浅复制的介绍请参考 3.1.3 节的介绍。

3.3.4 标准库 collections 中与字典有关的类

Python 标准库中提供了很多扩展功能,大幅度提高了开发效率。这里主要介绍 collections 中 OrderedDict 类、defaultdict 类和 Counter 类,deque、namedtuple 以及其他更多的类将在后面章节中进行介绍。

1. OrderedDict 类

Python 内置字典 dict 是无序的,如果需要一个可以记住元素插入顺序的字典,可以使用 collections. OrderedDict。

```
>>>import collections
>>>x=collections.OrderedDict()             #有序字典
>>>x['a']=3
>>>x['b']=5
>>>x['c']=8
>>>x
OrderedDict([('a',3),('b',5),('c',8)])
```

2. defaultdict 类

前面 3.3.2 节中字母出现频次统计的问题,也可以使用 collections 模块的 defaultdict 类来实现。

```
>>>import string
>>>import random
>>>x=string.ascii_letters+string.digits+string.punctuation
>>>z=''.join([random.choice(x) for i in range(1000)])
>>>from collections import defaultdict
>>>frequences=defaultdict(int)             #所有值默认为 0
>>>frequences
defaultdict(<class 'int'>,{})
>>>for item in z:
    frequences[item]+=1                    #修改每个字符的频次
>>>frequences.items()
```

创建 defaultdict 对象时,传递的参数表示字典中值的类型,除了上面代码演示的 int

类型，还可以是任意合法的 Python 类型。

```
>>>from collections import defaultdict
>>>games =defaultdict(list)                    #使用 list 作为值类型
>>>games                                       #所有值默认为空列表
defaultdict(<class 'list'>,{})
>>>games['name'].append('dong')                #可直接为字典 games 添加元素
>>>games['name'].append('zhang')
>>>games['score'].append(90)
>>>games['score'].append(93)
>>>games
defaultdict(<class 'list'>,{'score': [90,93],'name': ['dong','zhang']})
```

3. Counter 类

对于频次统计的问题，使用 collections 模块的 Counter 类可以更加快速地实现这个功能，并且能够提供更多的功能，例如，查找出现次数最多的元素。

```
>>>from collections import Counter
>>>frequences =Counter(z)                      #这里的 z 还是前面代码中的字符串对象
>>>frequences.items()
>>>frequences.most_common(1)                   #返回出现次数最多的一个字符及其频率
>>>frequences.most_common(3)                   #返回出现次数最多的前 3 个字符及其频率
```

3.4　集合：元素之间不允许重复

集合(set)属于 Python 无序可变序列，使用一对大括号作为定界符，元素之间使用逗号分隔，同一个集合内的每个元素都是唯一的，**元素之间不允许重复**。

集合中只能包含数字、字符串、元组等不可变类型(或者说可哈希)的数据，而不能包含列表、字典、集合等可变类型的数据。Python 提供了一个内置函数 hash()来计算对象的哈希值，凡是无法计算哈希值(调用内置函数 hash()时抛出异常)的对象都不能作为集合的元素，也不能作为字典对象的“键”。

3.4.1　集合对象的创建与删除

直接将集合赋值给变量即可创建一个集合对象。

```
>>>a ={3,5}                                    #创建集合对象
>>>type(a)                                     #查看对象类型
<class 'set'>
```

也可以使用 set()函数将列表、元组、字符串、range 对象等其他可迭代对象转换为集合，如果原来的数据中存在重复元素，则在转换为集合的时候只保留一个；如果原序列或迭代对象中有不可哈希的值，无法转换成为集合，抛出异常。

```
>>>a_set=set(range(8,14))                        #把 range 对象转换为集合
>>>a_set
{8,9,10,11,12,13}
>>>b_set=set([0,1,2,3,0,1,2,3,7,8])              #转换时自动去掉重复元素
>>>b_set
{0,1,2,3,7,8}
>>>x=set()                                       #空集合
```

除了列表推导式、生成器推导式、字典推导式之外,Python 还支持使用集合推导式来快速生成集合。

```
>>>{x.strip() for x in (' he ','she  ','  I')}
{'I','she','he'}
>>>import random
>>>x={random.randint(1,500) for i in range(100)}
                                                 #生成随机数,自动去除重复元素
>>>len(x)                                        #一般而言输出结果会小于 100
>>>{str(x) for x in range(10)}
{'3','0','1','8','4','7','5','6','9','2'}
```

当不再使用某个集合时,可以使用 del 命令删除整个集合。

3.4.2 集合操作与运算

1. 集合元素增加与删除

集合对象的 add( ) 方法可以增加新元素,如果该元素已存在则忽略该操作,不会抛出异常;update( ) 方法合并另外一个集合中的元素到当前集合中,并自动去除重复元素。

```
>>>s={1,2,3}
>>>s.add(3)                                      #添加元素,重复元素自动忽略
>>>s.update({3,4})                               #更新当前字典,自动忽略重复的元素
>>>s
{1,2,3,4}
```

集合对象的 pop( ) 方法随机删除并返回集合中的一个元素,如果集合为空则抛出异常;remove( ) 方法删除集合中的元素,如果指定元素不存在则抛出异常;discard( ) 方法从集合中删除一个特定元素,如果元素不在集合中则忽略该操作;clear( ) 方法清空集合。

```
>>>s.discard(5)                                  #删除元素,不存在则忽略该操作
>>>s.remove(5)                                   #删除元素,不存在就抛出异常
KeyError: 5
>>>s.pop()                                       #删除并返回一个元素
1
```

2. 集合运算

内置函数 len( ) 、max( ) 、min( ) 、sum( ) 、sorted( ) 、map( ) 、filter( ) 、enumerate( ) 等也适



用于集合。另外，Python 集合还支持数学意义上的交集、并集、差集等运算。

```
>>>a_set=set([8,9,10,11,12,13])
>>>b_set={0,1,2,3,7,8}
>>>a_set | b_set                          #并集
{0,1,2,3,7,8,9,10,11,12,13}
>>>a_set.union(b_set)                     #并集
{0,1,2,3,7,8,9,10,11,12,13}
>>>a_set & b_set                          #交集
{8}
>>>a_set.intersection(b_set)              #交集
{8}
>>>a_set.difference(b_set)                #差集
{9,10,11,12,13}
>>>a_set-b_set
{9,10,11,12,13}
>>>a_set.symmetric_difference(b_set)      #对称差集
{0,1,2,3,7,9,10,11,12,13}
>>>a_set ^ b_set
{0,1,2,3,7,9,10,11,12,13}
>>>x={1,2,3}
>>>y={1,2,5}
>>>z={1,2,3,4}
>>>x<y                                    #比较集合大小/包含关系
False
>>>x<z                                    #真子集
True
>>>y<z
False
>>>{1,2,3}<={1,2,3}                       #子集
True
>>>x.issubset(y)                          #测试是否为子集
False
>>>x.issubset(z)
True
>>>{3} & {4}
set()
>>>{3}.isdisjoint({4})                    #如果两个集合的交集为空,返回 True
True
```

需要注意的是，关系运算符 >、>=、<、<= 作用于集合时表示集合之间的包含关系，而不是比较集合中元素的大小关系。对于两个集合 A 和 B，如果 A<B 不成立，不代表 A>=B 就一定成立。

3.4.3 不可变集合 frozenset

Python 内置支持 frozenset 类，用法与 set 类基本相似，支持交集、并集、差集等运算以及测试是否为子集或超集等运算。与 set 类不同的是，frozenset 是不可变集合，没有提供 add()、remove()等可以修改集合对象的方法。

```
>>>x=frozenset(range(5))                    #创建不可变集合
>>>x
frozenset({0,1,2,3,4})
>>>x.add(5)                                 #不支持 add()方法,抛出异常
AttributeError: 'frozenset' object has no attribute 'add'
>>>x | frozenset(range(5,10))               #并集运算
frozenset({0,1,2,3,4,5,6,7,8,9})
>>>x & frozenset(range(5,10))               #交集运算
frozenset()
>>>x -frozenset(range(5,10))                #差集运算
frozenset({0,1,2,3,4})
>>>frozenset(range(4)) <frozenset(range(5))  #集合包含关系比较
True
```

3.4.4 集合应用案例

The Zen of Python 认为 There should be one—and preferably only one—obvious way to do it。编写代码时除了要准确地实现功能之外，还要考虑代码的优化，尽量找到一种更快、更好的方法实现预定功能。Python 字典和集合都使用 hash 表来存储元素，元素查找速度非常快，关键字 in 作用于字典和集合时比作用于列表要快得多。

```
import random
import time

x1 =list(range(10000))
x2 =tuple(range(10000))
x3 =set(range(10000))
x4 =dict(zip(range(10000),range(10000)))
r =random.randint(0,9999)

for t in (x4,x3,x2,x1):
    start =time.time()
    for i in range(9999999):
        r in t
    print(type(t),'time used:',time.time() -start)
```

从下面的运行结果可以看出，对于成员测试运算符 in，列表的效率远远不如字典和集



合，并且随着序列的变长，列表的查找速度越来越慢，而字典和集合基本上不受影响。

```
<class 'dict'>time used: 1.1570661067962646
<class 'set'>time used: 1.442082405090332
<class 'tuple'>time used: 1185.4768052101135
<class 'list'>time used: 1183.18967461586
```

作为集合的具体应用，可以使用集合快速提取序列中单一元素，即提取出序列中所有不重复的元素。如果使用传统方式的话，需要编写下面的代码：

```
>>>import random
#生成 100 个介于 0～9999 之间的随机数
>>>listRandom=[random.choice(range(10000)) for i in range(100)]
>>>noRepeat=[]
>>>for i in listRandom :
    if i not in noRepeat :
        noRepeat.append(i)
```

而如果使用集合的话，只需要下面这么一行代码就可以了。

```
>>>newSet=set(listRandom)
```

集合中的元素不允许重复，Python 集合的内部实现为此做了大量相应的优化，添加元素时如果已经存在则自动忽略。下面的代码用于返回指定范围内一定数量的不重复数字。

```
import random

def randomNumbers(number,start,end):
    '''使用集合来生成 number 个介于 start 和 end 之间的不重复随机数'''
    data=set()
    while len(data)<number:
        element=random.randint(start,end)
        data.add(element)
    return data
```

当然，如果在项目中需要这样一个功能的时候，还是直接使用 random 模块的 sample( ) 函数更好一些。但 random 模块的 sample( ) 函数只支持列表、元组、集合、字符串和 range 对象，不支持字典以及 map、zip、enumerate、filter 等惰性求值的迭代对象。

```
>>>import random
>>>random.sample(range(1000),20)                    #在指定分布中选取不重复元素
[61,538,873,815,708,609,995,64,7,719,922,859,807,464,789,651,31,702,504,25]
```

下面的两段代码用来测试指定列表中是否包含非法数据，很明显第二段使用集合的代码更高效一些。

```
import random
```



```
lstColor=('red','green','blue')
colors=[random.choice(lstColor) for i in range(10000)]

for item in colors:                                  #遍历列表中的元素并逐个判断
    if item not in lstColor:
        print('error:',item)
        break

if (set(colors)-set(lstColor)):                      #转换为集合之后再比较
    print('error')
```

下面的代码使用字典和集合模拟了有向图结构,并实现了节点的入度和出度计算,代码中所用的有向图如图 3-3 所示。

```
def getDegrees(orientedGraph,node):
    outDegree=len(orientedGraph.get(node,[]))
    inDegree=sum(1 for v in orientedGraph.values() if node in v)
    return (inDegree,outDegree)

graph={'a':set('bcdef'),'b':set('ce'),'c':set('d'),'d':set('e'),'e':set('f'),
       'f':set('cgh'),'g':set('fhi'),'h':set('fgi'),'i':set()}

print(getDegrees(graph,'h'))
```

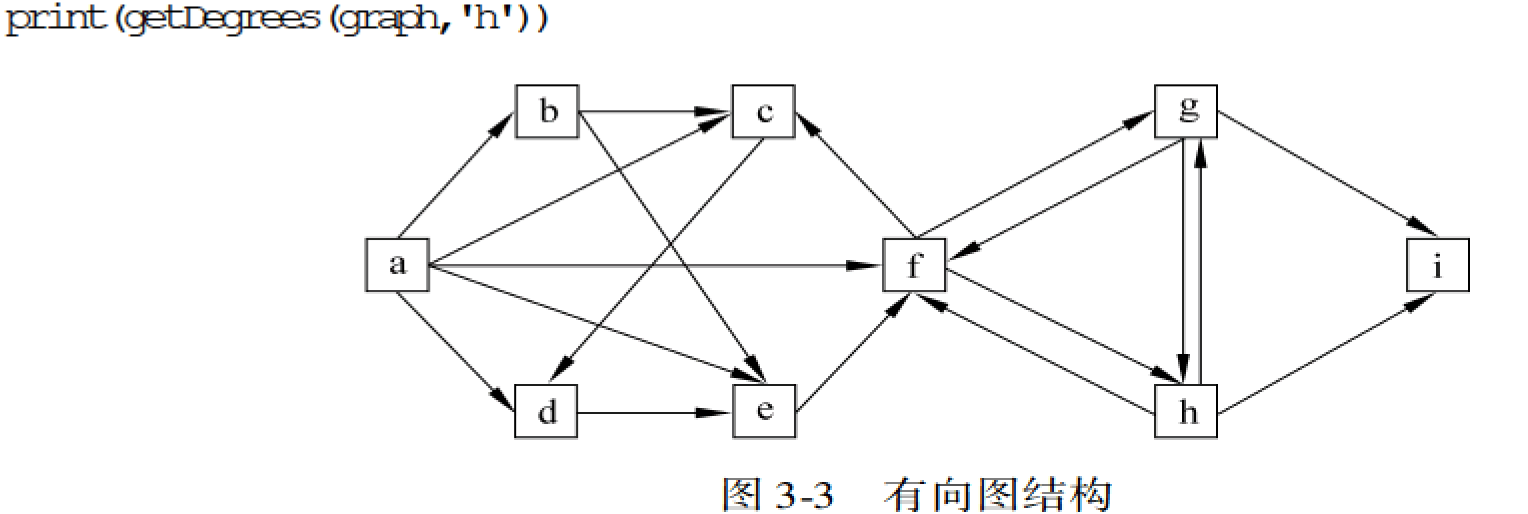


图 3-3 有向图结构

3.5 序列解包的多种形式和用法

序列解包(Sequence Unpacking)是 Python 中非常重要和常用的一个功能,可以使用非常简洁的形式完成复杂的功能,提高了代码的可读性,减少了程序员的代码输入量。

```
>>>x,y,z=1,2,3                                       #多个变量同时赋值
>>>v_tuple=(False,3.5,'exp')
>>>(x,y,z)=v_tuple
>>>x,y,z=v_tuple
>>>x,y,z=range(3)                                    #可以对 range 对象进行序列解包
>>>x,y,z=iter([1,2,3])                               #使用迭代器对象进行序列解包
>>>x,y,z=map(str,range(3))                           #使用可迭代的 map 对象进行序列解包
```




```
>>>a,b=b,a                                   #交换两个变量的值
```

序列解包还可以用于列表、字典、enumerate 对象、filter 对象、zip 对象等。对字典使用时，默认是对字典"键"进行操作，如果对"键:值"对进行操作应使用字典的 items()方法说明，如果需要对字典"值"进行操作应使用字典的 values()方法明确指定。

```
>>>a=[1,2,3]
>>>b,c,d=a                                   #列表也支持序列解包的用法
>>>x,y,z=sorted([1,3,2])                     #sorted()函数返回排序后的列表
>>>s={'a':1,'b':2,'c':3}
>>>b,c,d=s.items()                           #这里的重点是序列解包的用法
>>>b
('c',3)
>>>b,c,d=s                                   #使用字典时不用太多考虑元素的顺序
>>>b                                         #在 Python 3.6.x 和更新版本中略有不同是正常的
'c'
>>>b,c,d=s.values()
>>>print(b,c,d)
1 3 2
>>>a,b,c='ABC'                               #字符串也支持序列解包
>>>print(a,b,c)
A B C
```

使用序列解包可以很方便地同时遍历多个序列。

```
>>>keys=['a','b','c','d']
>>>values=[1,2,3,4]
>>>for k,v in zip(keys,values):
    print((k,v),end=' ')
('a',1) ('b',2) ('c',3) ('d',4)
>>>x=['a','b','c']
>>>for i,v in enumerate(x):
    print('The value on position {0} is {1}'.format(i,v))
The value on position 0 is a
The value on position 1 is b
The value on position 2 is c
>>>s={'a':1,'b':2,'c':3}
>>>for k,v in s.items():                     #字典中每个元素包含"键"和"值"两部分
    print((k,v),end=' ')
('a',1) ('b',2) ('c',3)
```

下面的代码演示了序列解包的另类用法和错误的用法：

```
>>>print(*[1,2,3],4,*(5,6))
1 2 3 4 5 6
>>>*range(4),4
```

```
(0,1,2,3,4)
>>>*range(4)                                    #不允许这样用
SyntaxError: can't use starred expression here
>>>{*range(4),4,*(5,6,7)}
{0,1,2,3,4,5,6,7}
>>>{'x':1,**{'y':2}}
{'y':2,'x':1}
>>>a,b,c=range(3)
>>>a,b,c=*range(3)                              #不允许这样用
SyntaxError: can't use starred expression here
>>>a,b,c,d=*range(3),3
```

下面的代码看起来与序列解包类似,但严格来说是序列解包的逆运算,与函数的可变长度参数一样(详见 5.2.4 节),用来收集等号右侧的多个数值。

```
>>>a,*b,c=1,2,3,4,5
>>>a,b,c
(1,[2,3,4],5)
>>>b
[2,3,4]
>>>a,*b,c=1,2,3,4
>>>a,b,c
(1,[2,3],4)
>>>a,*b,c=tuple(range(20))
>>>b
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18]
>>>*b=1,2,3,4                                   #等号左侧必须为列表、元组或多个变量
SyntaxError: starred assignment target must be in a list or tuple
```

3.6 标准库中的其他常用数据类型

除了大量内置数据类型,Python 还通过 collections、enum、array、heapq、fractions 等标准库提供了其他丰富的类型。

3.6.1 枚举类型

```
>>>from enum import Enum                        #导入模块中的类
>>>class Color(Enum):                           #创建自定义枚举类
    red=1
    blue=2
    green=3
>>>Color.red                                    #访问枚举类的成员
<Color.red: 1>
>>>type(Color.green)                            #查看枚举类成员的类型
```




```
 <enum 'Color'>
 >>>isinstance(Color.red,Color)
True
 >>>x=dict()
 >>>x[Color.red]='red'                        #枚举类成员可哈希，可以作为字典的“键”
 >>>x
{<Color.red: 1>: 'red'}
 >>>Color(2)                                   #返回指定值对应的枚举类成员
 <Color.blue: 2>
 >>>Color['red']
 <Color.red: 1>
 >>>r=Color.red
 >>>r.name
'red'
 >>>r.value
1
 >>>list(Color)                                #枚举类是可以迭代的
[<Color.red: 1>, <Color.blue: 2>, <Color.green: 3>]
```

3.6.2　数组类型

标准库 array 提供的 array 类支持数组的创建与使用，可以创建的数组类型包括整数、实数、Unicode 字符等，可以使用 help() 函数查看更完整的类型列表。

```
 >>>from array import array
 >>>s="Hello world"
 >>>sa=array('u',s)                            #创建可变字符串对象
 >>>print(sa)
array('u','Hello world')
 >>>print(sa.tostring())                       #查看可变字符串对象内容
b'H\x00e\x00l\x00l\x00o\x00 \x00w\x00o\x00r\x00l\x00d\x00'
 >>>print(sa.tounicode())                      #查看可变字符串对象内容
Hello world
 >>>sa[0]='F'                                  #修改指定位置上的字符
 >>>print(sa)
array('u','Fello world')
 >>>sa.insert(5,'w')                           #在指定位置插入字符
 >>>print(sa)
array('u','Fellow world')
 >>>sa.remove('l')                             #删除指定字符的首次出现
 >>>print(sa)
array('u','Felow world')
 >>>sa.remove('w')
 >>>print(sa)
```



```
array('u','Felo world')
>>>ia =array('I')                          #创建整型数组
>>>for i in range(5):
    ia.append(i)
>>>ia
array('I',[0,1,2,3,4])
>>>ia[0] =5
>>>ia
array('I',[5,1,2,3,4])
```

3.6.3 队 列

Python 标准库 queue 提供了 LILO 队列类 Queue、LIFO 队列类 LifoQueue、优先级队列类 PriorityQueue，标准库 collections 提供了双端队列。例如：

```
>>>from queue import Queue                 #LILO 队列
>>>q=Queue()                               #创建队列对象
>>>q.put(0)                                #在队列尾部插入元素
>>>q.put(1)
>>>q.put(2)
>>>print(q.queue)                          #查看队列中的所有元素
deque([0,1,2])
>>>q.get()                                 #返回并删除队列的头部元素
0
>>>q.get()
1
>>>q.get()
2

>>>from queue import LifoQueue             #LIFO 队列
>>>q=LifoQueue()                           #创建 LIFO 队列对象
>>>q.put(1)                                #在队列尾部插入元素
>>>q.put(2)
>>>q.put(3)
>>>q.queue                                 #查看队列中的所有元素
[1,2,3]
>>>q.get()                                 #返回并删除队列尾部元素
3
>>>q.get()
2
>>>q.queue
[1]
>>>q.get()                                 #对空队列调用 get()方法会阻塞当前线程
```

```
>>>from queue import PriorityQueue          #优先级队列
>>>q=PriorityQueue()                        #创建优先级队列对象
>>>q.put(3)                                 #插入元素
>>>q.put(8)                                 #插入元素
>>>q.put(100)
>>>q.queue                                  #查看优先级队列中的所有元素
[3,8,100]
>>>q.put(1)                                 #插入元素,自动调整优先级队列
>>>q.put(2)
>>>q.queue
[1,2,100,8,3]
>>>q.get()                                  #返回并删除优先级最低的元素
1
>>>q.get()                                  #多执行几次该语句并观察返回的数据
2

>>>from collections import deque
>>>q=deque(maxlen=5)                        #创建双端队列
>>>for item in [3,5,7,9,11]:                #添加元素
    q.append(item)
>>>q.append(13)                             #队列满,自动溢出
>>>q.append(15)
>>>q
deque([7,9,11,13,15],maxlen=5)
>>>q.appendleft(5)                          #从左侧添加元素,右侧自动溢出
>>>q
deque([5,7,9,11,13],maxlen=5)
>>>q.popleft()                              #弹出并返回最左端元素
5
>>>q.pop()                                  #弹出并返回最右端元素
13
>>>q
deque([7,9,11],maxlen=5)
>>>q.insert(2,10)                           #在中间位置插入元素
>>>q
deque([7,9,10,11],maxlen=5)
>>>q+=[1,2]                                 #追加多个元素
>>>q
deque([9,10,11,1,2],maxlen=5)
>>>q.pop()
2
>>>q.pop()
1
>>>q
```

```
deque([9,10,11],maxlen=5)
>>>q*=2                                          #序列重复
>>>q
deque([10,11,9,10,11],maxlen=5)
>>>q.count(10)                                   #返回元素的出现次数
2
>>>q.rotate(2)                                   #循环右移 2 个元素
>>>q
deque([10,11,10,11,9],maxlen=5)
>>>q.rotate(-2)                                  #循环左移 2 个元素
>>>q
deque([10,11,9,10,11],maxlen=5)
```

3.6.4 具名元组

```
>>>from collections import namedtuple
>>>Point=namedtuple('Point',['x','y','z'])       #创建具名元组类
>>>Point
<class '__main__.Point'>
>>>p=Point(3,4,5)                                #实例化对象
>>>p
Point(x=3,y=4,z=5)
>>>p.x                                           #访问成员
3
>>>p._fields                                     #查看字段列表
('x','y','z')
>>>p._replace(x=30,z=8)                          #替换成员值,返回新对象
Point(x=30,y=4,z=8)
>>>p.x=7                                         #不允许这样直接赋值
AttributeError: can't set attribute
>>>d=dict()
>>>d[p]='spirit position'                        #具名元组对象可以作为字典的"键"
>>>d
{Point(x=3,y=4,z=5): 'spirit position'}
```

3.6.5 堆

堆是一个特殊的二叉树,其中每个父节点的值都小于或等于其所有子节点的值。使用数组或列表来实现小根堆时,对于所有的 k(下标,从 0 开始)都满足 heap[k] <= heap[2*k+1] 和 heap[k] <= heap[2*k+2],并且整个堆中最小的元素总是位于二叉树的根节点,大根堆与小根堆正好相反。Python 在 heapq 模块中提供了对堆的支持。

```
>>>import heapq                                  #导入 heapq 模块
```

```
>>>import random
>>>data =random.sample(range(1000),10)              #生成随机测试数据
>>>data
[638,659,212,84,737,677,553,340,526,747]
>>>heapq.heapify(data)                              #堆化随机测试数据
>>>data
[84,340,212,526,737,677,553,659,638,747]
>>>heapq.heappush(data,30)                          #新元素入堆，自动调整堆结构
>>>data
[30,84,212,526,340,677,553,659,638,747,737]
>>>heapq.heappush(data,5)
>>>data
[5,84,30,526,340,212,553,659,638,747,737,677]
>>>heapq.heappop(data)                              #返回并删除最小元素，自动调整堆
5
>>>heapq.heappop(data)
30
>>>heapq.heappop(data)
84
>>>data
[212,340,553,526,737,677,747,659,638]
>>>heapq.heappushpop(data,1000)                     #弹出最小元素，同时新元素入堆
212
>>>data
[340,526,553,638,737,677,747,659,1000]
>>>heapq.heapreplace(data,500)                      #弹出最小元素，同时新元素入堆
340
>>>data
[500,526,553,638,737,677,747,659,1000]
>>>heapq.heapreplace(data,700)
500
>>>data
[526,638,553,659,737,677,747,700,1000]
>>>heapq.nlargest(3,data)                           #返回最大的前 3 个元素
[1000,747,737]
>>>heapq.nsmallest(2,data,key =str)                 #返回指定排序规则下最小的 2 个元素
[1000,526]
```

第4章 反者，道之动：程序控制结构



有了合适的数据类型和数据结构之后，还要依赖于选择和循环结构来实现特定的业务逻辑。一个完整的选择结构或循环结构可以看作是一个大的“语句”，从这个角度来讲，程序中的多条“语句”是顺序执行的。

4.1 条件表达式

在选择结构和循环结构中，都要根据条件表达式的值来确定下一步的执行流程。条件表达式的值只要不是 False、0（或 0.0、0j 等）、空值 None、空列表、空元组、空集合、空字典、空字符串、空 range 对象或其他空迭代对象，Python 解释器均认为与 True 等价。从这个意义上来讲，所有的 Python 合法表达式都可以作为条件表达式，包括含有函数调用的表达式。

关于表达式和运算符的详细内容请参考 2.2 节，这里再重点介绍一下几个比较特殊的运算符。

1. 关系运算符

Python 中的关系运算符可以连续使用，这样不仅可以减少代码量，也比较符合人类的思维方式。

```
>>>print(1<2<3)                          #等价于 1<2 and 2<3
True
>>>print(1<2>3)
False
>>>print(1<3>2)
True
```

在 Python 语法中，条件表达式中不允许使用赋值运算符“=”，避免了误将关系运算符“==”写成赋值运算符“=”带来的麻烦。在条件表达式中使用赋值运算符“=”将抛出异常，提示语法错误。

```
>>>if a=3:                               #条件表达式中不允许使用赋值运算符
SyntaxError: invalid syntax
>>>if (a=3) and (b=4):
SyntaxError: invalid syntax
```

关系运算符具有惰性计算的特点，只计算必须计算的值，而不是计算关系表达式中的每个表达式。

```
>>>1 >2 >xxx                                    #当前上下文中并不存在变量 xxx
False
```

2. 逻辑运算符

逻辑运算符 and、or、not 分别表示与、或、非 3 种逻辑运算，在功能上可以与电路的连接方式做个简单类比：or 运算符类似于并联电路，只要有一个开关是通的那么灯就是亮的；and 运算符类似于串联电路，必须所有开关都是通的灯才会亮；not 运算符类似于短路电路，如果开关通了那么灯就灭了，如图 4-1 所示。

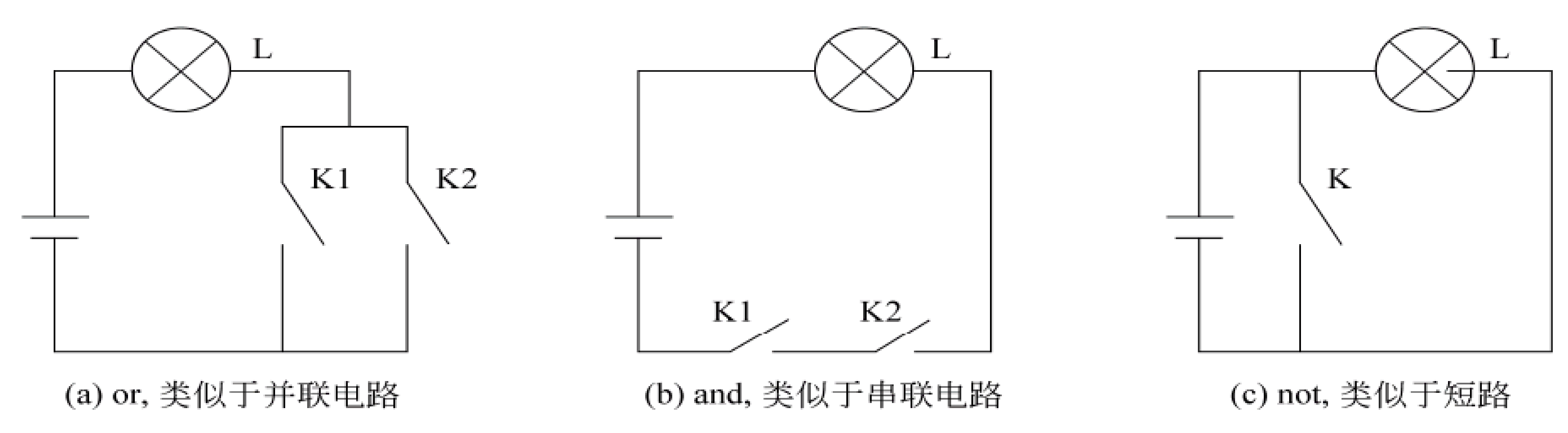


图 4-1　逻辑运算符与几种电路的类比关系

与关系运算符类似，逻辑运算符 and 和 or 具有短路求值或惰性求值的特点，可能不会对所有表达式进行求值，而是只计算必须计算的表达式的值。以 and 为例，对于表达式“表达式 1 and 表达式 2”而言，如果“表达式 1”的值为 False 或其他等价值时，不论“表达式 2”的值是什么，整个表达式的值都是 False，丝毫不受“表达式 2”的影响，因此“表达式 2”不会被计算。在设计包含多个条件的条件表达式时，如果能够大概预测不同条件失败的概率，并将多个条件根据 and 和 or 运算符的短路求值特性来组织顺序，可以提高程序运行效率。

```
>>>3 and 5
5
>>>3 or 5
3
>>>0 and 5
0
>>>0 or 5
5
>>>not 3
False
>>>not 0
True
```

下面的函数使用指定的分隔符把多个字符串连接成一个字符串，如果用户没有指定分隔符则使用逗号。

```
>>>def Join(chList,sep=None):
    return (sep or ',').join(chList)                    #注意：参数 sep 不是字符串时会抛出异常
>>>chTest=['1','2','3','4','5']
>>>Join(chTest)
'1,2,3,4,5'
>>>Join(chTest,':')
'1:2:3:4:5'
```

当然，也可以把上面的函数直接定义为下面带有默认值参数的形式：

```
>>>def Join(chList,sep=','):
    return sep.join(chList)
```

4.2 选择结构

常见的选择结构有单分支选择结构、双分支选择结构、多分支选择结构以及嵌套的分支结构，也可以构造跳转表来实现类似的逻辑。另外，循环结构和异常处理结构中也可以带有 else 子句，可以看作是特殊形式的选择结构，请参考 4.3 节和 11.1 节的介绍。

4.2.1 单分支选择结构

单分支选择结构语法如下所示，其中表达式后面的冒号“：”是不可缺少的，表示一个语句块的开始，并且语句块必须做相应地缩进，一般是以 4 个空格为缩进单位。

```
if 表达式:
    语句块
```

当表达式值为 True 或其他与 True 等价的值时，表示条件满足，语句块被执行，否则该语句块不被执行，而是继续执行后面的代码（如果有的话），如图 4-2 所示。

下面的代码演示了单分支选择结构的用法：

```
x=input('Input two numbers:')
a,b=map(int,x.split())
if a>b:
    a,b=b,a                                   #序列解包，交换两个变量的值
print(a,b)
```

在 Python 中，代码的缩进非常重要，**缩进是体现代码逻辑关系的重要方式，同一个代码块必须保证相同的缩进量**。在实际开发中，只要遵循一定的约定，Python 代码的排版是可以降低要求的，例如下面的代码，虽然不建议这样写，但确实是可以执行的。

```
>>>if 3>2: print('ok')                        #如果语句较短，可以直接写在分支语句后面
ok
```



```
>>>if True:print(3);print(5)                  #在一行写多个语句,使用分号分隔
3
5
```

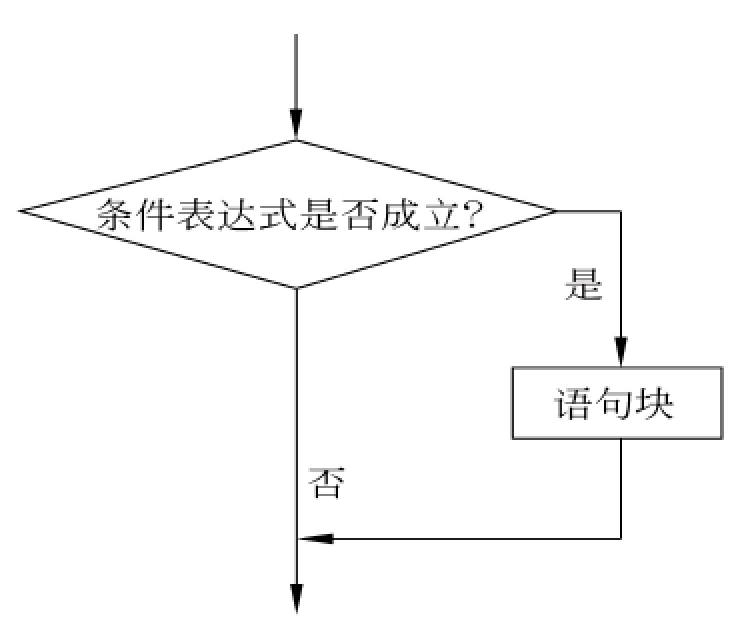


图 4-2　单分支选择结构

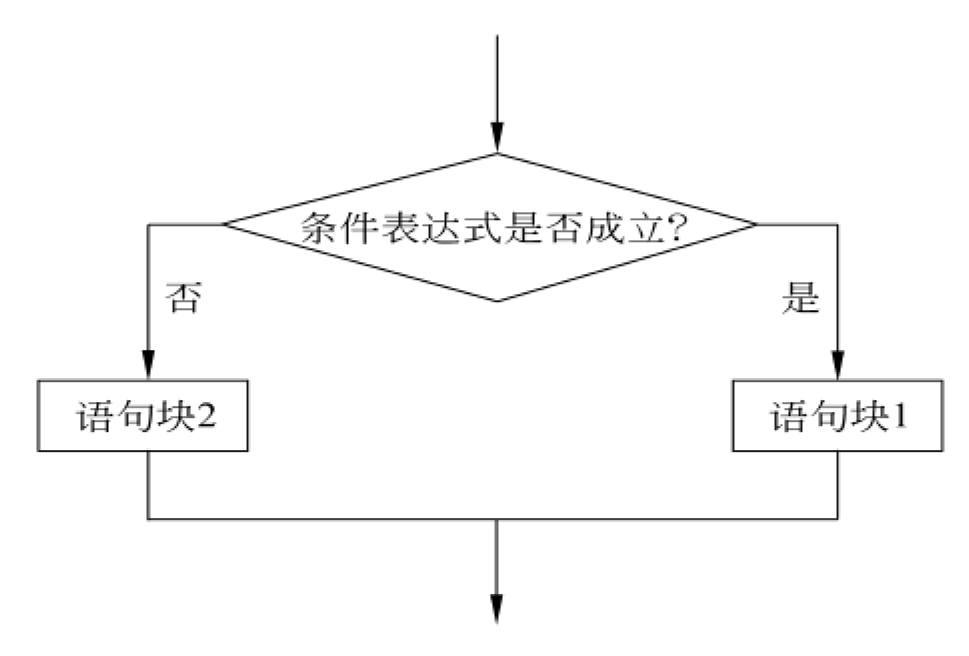


图 4-3　双分支选择结构

4.2.2　双分支选择结构

双分支选择结构的语法为

```
if 表达式:
    语句块 1
else:
    语句块 2
```

当表达式值为 True 或其他等价值时,执行语句块 1,否则执行语句块 2。语句块 1 或语句块 2 总有一个会执行,然后再执行后面的代码(如果有的话),如图 4-3 所示。

下面的代码通过鸡兔同笼问题演示了双分支结构的用法。

```
jitu,tui=map(int,input('请输入鸡兔总数和腿总数:').split())
tu=(tui -jitu*2)/2
if int(tu)==tu:
    print('鸡:{0},兔:{1}'.format(int(jitu-tu),int(tu)))
else:
    print('数据不正确,无解')
```

另外,Python 还提供了一个三元运算符,并且在三元运算符构成的表达式中还可以嵌套三元运算符,可以实现与选择结构相似的效果。语法为

```
value1 if condition else value2
```

当条件表达式 condition 的值与 True 等价时,表达式的值为 value1,否则表达式的值为 value2。另外,value1 和 value2 本身也可以是复杂表达式,也可以包含函数调用,甚至可以是三元运算符构成的表达式。这个结构的表达式也具有惰性求值的特点。

```
>>>a=5
>>>print(6) if a>3 else print(5)
6
>>>print(6 if a>3 else 5)                          #虽然结果与上一行代码一样，但代码含义不同
6
>>>b=6 if a>13 else 9                                          #赋值运算符的优先级非常低
>>>b
9
>>>x=math.sqrt(9) if 5>3 else random.randint(1,100)            #还没有导入 math 模块
NameError: name 'math' is not defined
>>>import math
>>>x=math.sqrt(9) if 5>3 else random.randint(1,100)            #还没有导入 random 模块
                                                               #但表达式 5>3 的值为 True
                                                               #所以可以正常运行
>>>x=math.sqrt(9) if 2>3 else random.randint(1,100)            #条件表达式 2>3 的值为 False
                                                               #需要计算第二个表达式
                                                               #但此时还没导入 random
                                                               #所以出错
NameError: name 'random' is not defined
>>>import random                                               #导入 random，成功执行
>>>x=math.sqrt(9) if 2>3 else random.randint(1,100)
```

虽然三元运算符可以嵌套使用，可以实现复杂的多分支选择结构的效果，但这样的代码可读性非常差，不建议使用。

```
>>>x=3
>>>(1 if x>2 else 0) if f(x)>5 else ('a' if x<5 else 'b')      #可以嵌套使用，建议不这样写
1
>>>x=0
>>>(1 if x>2 else 0) if f(x)>5 else ('a' if x<5 else 'b')
'a'
```

4.2.3 多分支选择结构

多分支选择结构的语法为

```
if 表达式 1:
      语句块 1
elif  表达式 2:
      语句块 2
elif  表达式 3:
      语句块 3
 ⋮
else:
      语句块 n
```

其中，关键字 elif 是 else if 的缩写。下面的代码演示了如何利用多分支选择结构将成绩从百分制变换到等级制。

```
def func(score):
    if score >100 or score <0:
        return 'wrong score.must between 0 and 100.'
    elif score  >=90:
        return 'A'
    elif score  >=80:
        return 'B'
    elif score  >=70:
        return 'C'
    elif score  >=60:
        return 'D'
    else:
        return 'E'
```

4.2.4 选择结构的嵌套

选择结构可以进行嵌套来实现复杂的业务逻辑，语法如下：

```
if 表达式 1:
    语句块 1
    if 表达式 2:
        语句块 2
    else:
        语句块 3
else:
    if 表达式 4:
        语句块 4
```

```
1 | if 表达式 1:
  | 2 | 语句块 1
  |   | if 表达式 2:
  |   |   3 | 语句块 2
  |   | else:
  |   |   3 | 语句块 3
  | else:
  | 2 | if 表达式 4:
  |   |   3 | 语句块 4
```

图 4-4 代码层次与隶属关系

上面语法示意中的代码层次和隶属关系如图 4-4 所示，注意相同层次的代码必须具有相同的缩进量。

使用嵌套选择结构时，一定要严格控制好不同级别代码块的缩进量，因为这决定了不同代码块的从属关系和业务逻辑是否被正确地实现，以及代码是否能够被解释器正确理解和执行。例如，前面百分制转等级制的代码，作为一种编程技巧，还可以尝试下面的写法：

```
def func(score):
    degree ='DCBAAE'
    if score >100 or score <0:
        return 'wrong.score must between 0 and 100.'
    else:
        index =(score  -60) // 10
```

```
        if index >=0:
            return degree[index]
        else:
            return degree[-1]
```

4.2.5 构建跳转表实现多分支选择结构

使用列表、元组或字典可以很容易构建跳转表,在某些场合下可以更快速地实现类似于多分支选择结构的功能。例如,下面的代码根据用户输入内容的不同来调用不同的函数完成不同的功能,其中就用到了字典形式的跳转表。

```
funcDict={'1':lambda:print('You input 1'),
          '2':lambda:print('You input 2'),
          '3':lambda:print('You input 3')}
x=input('Input an integer to call different function:')
func=funcDict.get(x,None)
if func:
    func()
else:
    print('Wrong integer.')
```

4.3 循环结构

4.3.1 for 循环与 while 循环

Python 主要有 for 循环和 while 循环两种形式的循环结构,多个循环可以嵌套使用,并且还经常和选择结构嵌套使用来实现复杂的业务逻辑。while 循环一般用于循环次数难以提前确定的情况,当然也可以用于循环次数确定的情况;for 循环一般用于循环次数可以提前确定的情况,尤其适用于枚举或遍历序列或迭代对象中元素的场合。对于带有 else 子句的循环结构,如果循环因为条件表达式不成立或序列遍历结束而**自然结束**时则执行 else 结构中的语句,如果循环是因为执行了 break 语句而导致循环**提前结束**则不会执行 else 中的语句。两种循环结构的完整语法形式分别为

```
while 条件表达式:
    循环体
[else:
    else 子句代码块]
```

和

```
for 取值 in 序列或迭代对象:
    循环体
[else:
    else 子句代码块]
```



其中，方括号内的 else 子句可以没有，也可以有。下面的代码使用循环结构遍历并输出列表中的所有元素。

```
a_list=['a','b','mpilgrim','z','example']
for i,v in enumerate(a_list):
    print('列表的第',i+1,'个元素是：',v)
```

下面的代码用来输出 1 ~ 100 之间能被 7 整除但不能同时被 5 整除的所有整数。

```
for i in range(1,101):
    if i%7==0 and i%5!=0:
        print(i)
```

下面的代码使用嵌套的循环结构打印九九乘法表。

```
for i in range(1,10):
    for j in range(1,i+1):
        print('{0}*{1}={2}'.format(i,j,i*j),end=' ')
    print()                          #打印空行
```

下面的代码演示了带有 else 子句的循环结构，该代码用来计算 1 +2 +3 +… +99 +100 的结果。

```
s=0
for i in range(1,101):               #不包括 101
    s+=i
else:
    print(s)
```

下面的代码使用 while 循环实现了同样的功能：

```
s=i=0
while i<=100:
    s+=i
    i+=1
else:
    print(s)
```

当然，上面的两段代码只是为了演示循环结构的用法，其中的 else 子句实际上并没有必要，循环结束后直接输出结果就可以了。另外，如果只是要计算 1 +2 +3 +… +99 +100 的值的话，直接用内置函数 sum( ) 和 range( ) 就可以了。

```
>>>sum(range(1,101))
5050
```

4.3.2　break 与 continue 语句

break 与 continue 语句在 while 循环和 for 循环中都可以使用，并且一般常与选择结构

或异常处理结构结合使用。一旦 break 语句被执行，将使得 break 语句所属层次的循环提前结束；continue 语句的作用是提前结束本次循环，忽略 continue 之后的所有语句，提前进入下一次循环。

下面的代码用来计算小于 100 的最大素数，内循环用来测试特定的整数 n 是否为素数，如果其中的 break 语句得到执行则说明 n 不是素数，并且由于循环提前结束而不会执行后面的 else 子句。如果某个整数 n 为素数，则内循环中的 break 语句不会执行，内循环自然结束后执行后面 else 子句中的语句，输出素数 n 之后执行 break 语句跳出外循环。

```
for n in range(100,1, -1):
    if n%2==0:
        continue
    for i in range(3,int(n**0.5)+1,2):
        if n%i==0:
            #结束内循环
            break
    else:
        print(n)
        #结束外循环
        break
```

需要注意的是，过多的 break 和 continue 语句会降低程序的可读性。因此，除非 break 或 continue 语句可以让代码更简单或更清晰，否则不要轻易使用。

4.3.3 循环代码优化技巧

实际开发中，正确实现了预定功能之后，一般还需要再优化一下代码以追求更高的执行效率。如果能从算法层面上进行优化，那毫无疑问会带来效率的大幅度提升。例如，判断一个大整数 n 是否为素数，如果根据素数定义去判断的话应该逐个测试[2,n-1]区间上的数是否能够整除 n，而实际上只需要判断从 2 到 n 的平方根这个小范围就可以了，再进一步说，实际上只需要判断 2 以及 3 到 n 的平方根之间所有奇数这个更小的范围。对于大整数 n 来说，循环次数和余数运算的次数减少是非常可观的，n 越大算法效率的提高越显著。

在编写循环语句时，应尽量减少循环内部不必要或无关的计算，与循环变量无关的代码应该尽可能地提取到循环之外。尤其是多重循环嵌套的情况，一定要尽量减少内层循环中不必要的计算，尽最大可能地把计算向外提。例如下面的代码，第二段明显比第一段的运行效率要高。

```
digits=(1,2,3,4)

for i in range(1000):
    result=[]
    for i in digits:
        for j in digits:
```

```
            for k in digits:
                result.append(i *100 +j *10 +k)

for i in range(1000):
    result =[]
    for i in digits:
        i =i *100
        for j in digits:
            j =j *10
            for k in digits:
                result.append(i +j +k)
```

另外，在循环中应尽量引用局部变量，局部变量的查询和访问速度比全局变量略快。同样的道理，在使用模块中的方法时，可以通过将其转换为局部变量来提高运行速度。例如下面的代码，第二段代码的速度就比第一段代码略快。当然，也可以使用 from math import sin as loc_sin 来代替其中的写法。

```
import math

for i in range(10000000):
    math.sin(i)

loc_sin =math.sin
for i in range(10000000):
    loc_sin(i)
```

代码优化涉及的内容非常广泛，除了在算法层面的优化之外，编码过程本身对程序员的功底要求也非常高。除了上面介绍的循环代码优化，本书其他章节中也会涉及一些优化的内容。例如，如果经常需要测试一个序列是否包含一个元素就应该尽量使用字典或集合而不使用列表，把多个字符串连接成一个字符串时尽量使用 join( ) 方法而不要使用运算符 +，对列表进行元素的插入和删除操作时应尽量从列表尾部进行，等等。实际开发中需要注意的是，首先要把代码写对，保证完全符合功能要求，然后再进行必要的优化来提高性能。过早地追求性能优化有时候可能会带来灾难而浪费大量精力。

4.4　精彩案例赏析

示例 4-1　输入若干个成绩，求所有成绩的平均分。每输入一个成绩后询问是否继续输入下一个成绩，回答 yes 就继续输入下一个成绩，回答 no 就停止输入成绩。

```
numbers =[]                                 #使用列表存放临时数据
while True:
    x =input('请输入一个成绩：')
    try:                                    #异常处理结构有关知识见第 11 章
        numbers.append(float(x))
```

```
    except:
        print('不是合法成绩')
    while True:
        flag=input('继续输入吗? (yes/no)')
        if flag.lower() not in ('yes','no'):        #限定用户输入内容必须为 yes 或 no
            print('只能输入 yes 或 no')
        else:
            break
    if flag.lower()=='no':
        break

print(sum(numbers)/len(numbers))
```

示例 4-2 编写程序,判断今天是今年的第几天。

```
import time

date=time.localtime()                                #获取当前日期时间
year,month,day=date[:3]
day_month=[31,28,31,30,31,30,31,31,30,31,30,31]
if year%400==0 or (year%4==0 and year%100!=0):     #判断是否为闰年
    day_month[1]=29
if month==1:
    print(day)
else:
    print(sum(day_month[:month-1])+day)
```

Python 标准库 datetime 提供了 datetime 和 timedelta 对象可以很方便地计算指定年、月、日、时、分、秒之前或之后的日期时间,还提供了返回结果中包含"今天是今年第几天""今天是本周第几天"等答案的 timetuple( ) 函数,等等。

```
>>>import datetime
>>>Today=datetime.date.today()
>>>Today
datetime.date(2016,10,8)
>>>Today -datetime.date(Today.year,1,1) +datetime.timedelta(days=1)
datetime.timedelta(282)
>>>Today.timetuple().tm_yday                         #今天是今年的第几天
282
>>>Today.replace(year=2013)                          #替换日期中的年
datetime.date(2013,10,8)
>>>Today.replace(month=1)                            #替换日期中的月
datetime.date(2016,1,8)
>>>now=datetime.datetime.now()
>>>now
datetime.datetime(2016,10,8,15,55,16,272174)
>>>now.replace(second=30)                            #替换日期时间中的秒
```

```
datetime.datetime(2016,10,8,15,55,30,272174)
>>>now +datetime.timedelta(days=5)                #计算 5 天后的日期时间
datetime.datetime(2016,10,13,15,55,16,272174)
>>>now +datetime.timedelta(weeks=-5)              #计算 5 周前的日期时间
datetime.datetime(2016,9,3,15,55,16,272174)
>>>def daysBetween(year1,month1,day1,year2,month2,day2):
    from datetime import date                     #计算两个日期之间相差多少天
    dif=date(year1,month1,day1) -date(year2,month2,day2)
    return dif.days

>>>daysBetween(2016,12,11,2016,11,27)
14
>>>daysBetween(2016,12,11,2011,11,27)
1841
```

另外,标准库 calendar 也提供了一些与日期操作有关的方法。例如:

```
>>>import calendar                                #导入模块
>>>print(calendar.calendar(2016))                 #查看 2016 年的日历表,结果略
>>>print(calendar.month(2016,11))                 #查看 2016 年 11 月份的日历表
>>>calendar.isleap(2016)                          #判断是否为闰年
True
>>>calendar.weekday(2016,10,26)                   #查看指定日期是周几
2
```

当然,也可以自己编写代码模拟 Python 标准库 calendar 中查看日历的方法。

```
from datetime import date

daysOfMonth=[31,28,31,30,31,30,31,31,30,31,30,31]

def myCalendar(year,month):
    #获取 year 年 month 月 1 日是周几
    start=date(year,month,1).timetuple().tm_wday
    #打印头部信息
    print('{0}年{1}月日历'.format(year,month).center(56))
    print('日\t一\t二\t三\t四\t五\t六')
    #获取该月有多少天,如果是 2 月并且是闰年,适当调整一下
    day=daysOfMonth[month-1]
    if month==2:
        if year%400==0 or (year%4==0 and year%100!=0):
            day +=1
    #生成数据,根据需要在前面填充空白
    result=[' '*8 for i in range(start+1)]
    result +=list(map(lambda d: str(d).ljust(8),range(1,day+1)))
    #打印数据
    for i,day in enumerate(result):
```

```
            if i!=0 and i%7==0:
                print()
            print(day,end='')
        print()

def main(year,month=-1):
    if type(year)!=int or year<1000 or year>10000:
        print('Year error')
        return
    if type(month)==int:
        #如果没有指定月份,就打印全年的日历
        if month==-1:
            for m in range(1,13):
                myCalendar(year,m)
        #如果指定了月份,就只打印这一个月的日历
        elif month in range(1,13):
            myCalendar(year,month)
        else:
            print('Month error')
            return
    else:
        print('Month error')
        return

main(2017)
```

示例 4-3 编写代码,输出由星号 * 组成的菱形图案,并且可以灵活控制图案的大小。

```
def main(n):
    for i in range(n):
        print(('*  '*i).center(n*3))
    for i in range(n,0,-1):
        print(('*  '*i).center(n*3))
```

图 4-5 和图 4-6 分别为参数 n=6 和 n=10 的运行效果。

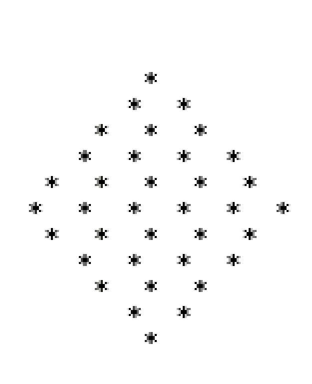


图 4-5 n=6 的运行效果

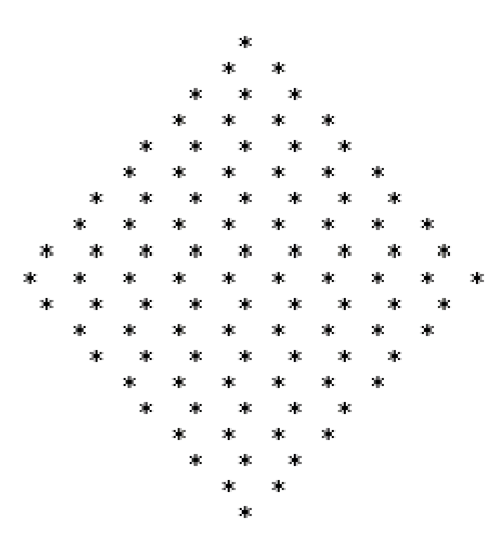


图 4-6 n=10 的运行效果



示例 4-4　快速判断一个数是否为素数。

```
n=input("Input an integer:")
n=int(n)
if n==2:
    print('Yes')
#偶数必然不是素数
elif n%2==0:
    print('No')
else:
    #大于 5 的素数必然出现在 6 的倍数两侧
    #因为 6x+2、6x+3、6x+4 肯定不是素数,假设 x 为大于 1 的自然数
    m=n%6
    if m!=1 and m!=5:
        print('No')
    else:
        for i in range(3,int(n**0.5)+1,2):
            if n%i==0:
                print('No')
                break
        else:
            print('Yes')
```

示例 4-5　编写程序，计算组合数 C(n,i)，即从 n 个元素中任选 i 个，有多少种选法。根据组合数定义，可以编写代码如下：

```
import math

def Cni1(n,i):
    return int(math.factorial(n)/math.factorial(i)/math.factorial(n-i))
```

虽然在 Python 中不用担心数字太大而超过变量的表示范围，但是计算大整数的阶乘也确实需要一些时间，尤其是上面的函数中存在大量的重复计算，严重影响速度。如果把组合数的定义展开并化简一下的话可以发现其中隐藏的规律，以 Cni(8,3) 为例，Cni(8,3) =8!/3!/(8-3)! =(8×7×6×5×4×3×2×1)/(3×2×1)/(5×4×3×2×1)，对于(5,8]区间的数，分子上出现一次而分母上没出现；(3,5]区间的数在分子、分母上各出现一次；[1,3]区间的数分子上出现一次而分母上出现两次。根据这一规律，可以编写如下非常高效的组合数计算程序。

```
def Cni2(n,i):
    if not (isinstance(n,int) and isinstance(i,int) and n>=i):
        print('n and i must be integers and n must be larger than or equal to i.')
        return
    result=1
    Min,Max=sorted((i,n-i))
```

```
    for i in range(n,0,-1):
        if i>Max:
            result*=i
        elif i<=Min:
            result/=i
    return result

print(Cni2(6,2))
```

Python 标准库 itertools 提供了组合函数 combinations( )、排列函数 permutations( )、用于循环遍历可迭代对象元素的函数 cycle( )、根据一个序列的值对另一个序列进行过滤的函数 compress( )、根据函数返回值对序列进行分组的函数 groupby( )、返回包含无限连续值的 count 对象的 count 函数( )、计算笛卡儿积的函数 product( )等。下面的代码演示了部分函数的用法。

```
>>>import itertools
>>>for it in itertools.combinations(range(1,5),3):      #从 4 个元素中选 3 个元素的组合
    print(it,end=' ')
(1,2,3) (1,2,4) (1,3,4) (2,3,4)
>>>list(itertools.permutations([1,2,3,4],3))           #从 4 个元素中任选 3 个元素的排列
>>>x=itertools.permutations([1,2,3,4],4)               #4 个元素的全排列
>>>for i in range(5):                                  #输出前 5 个排列
    print(next(x),end=' ')
(1,2,3,4) (1,2,4,3) (1,3,2,4) (1,3,4,2) (1,4,2,3)
>>>x='Private Key'
>>>y=itertools.cycle(x)                                #循环遍历序列中的元素
>>>for i in range(20):
    print(next(y),end=',')
P,r,i,v,a,t,e, ,K,e,y,P,r,i,v,a,t,e, ,K,
>>>for i in range(5):
    print(next(y),end=',')
e,y,P,r,i,
>>>x=range(1,20)
>>>y=(1,0)*9+(1,)
>>>y
(1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1)
>>>list(itertools.compress(x,y))                       #根据一个序列的值,对另一个序列进行过滤
[1,3,5,7,9,11,13,15,17,19]
>>>def group(v):
    if v>10:
        return 'greater than 10'
    elif v<5:
        return 'less than 5'
    else:
```

```
            return 'between 5 and 10'
>>>x=range(20)
>>>y=itertools.groupby(x,group)                          #对序列元素进行分组
>>>for k,v in y:
    print(k,':',list(v))
less than 5 : [0,1,2,3,4]
between 5 and 10 : [5,6,7,8,9,10]
greater than 10 : [11,12,13,14,15,16,17,18,19]
>>>x=itertools.count(5,3)                                #起始值为 5、步长为 3 的 count 对象
>>>for i in range(10):
    print(next(x),end=' ')
5 8 11 14 17 20 23 26 29 32
>>>list(zip('abcde',itertools.count()))                  #count 对象中的元素个数是无穷的
[('a',0),('b',1),('c',2),('d',3),('e',4)]
>>>list(zip('abc',itertools.count()))
[('a',0),('b',1),('c',2)]
>>>list(zip('abcdefghi',itertools.count()))
[('a',0),('b',1),('c',2),('d',3),('e',4),('f',5),('g',6),('h',7),('i',8)]
>>>for item in itertools.product('abc',range(4)):        #笛卡儿积
    print(item,end=' ')
('a',0) ('a',1) ('a',2) ('a',3) ('b',0) ('b',1) ('b',2) ('b',3) ('c',0) ('c',1)('c',2) ('c',3)
>>>for item in itertools.product('abc','123','BC'):
    print(item,end=' ')
('a','1','B') ('a','1','C') ('a','2','B') ('a','2','C') ('a','3','B') ('a','3','C') ('b','1','B') ('b
','1','C') ('b','2','B') ('b','2','C') ('b','3','B') ('b','3','C') ('c','1','B') ('c','1','C') ('c','
2','B') ('c','2','C') ('c','3','B')('c','3','C')
>>>func=lambda x:x.isnumeric()
>>>list(itertools.takewhile(func,'1234abcd'))            #过滤元素
['1','2','3','4']
>>>list(itertools.dropwhile(func,'1234abcd'))
['a','b','c','d']
```

示例 4-6　编写代码，模拟决赛现场最终成绩的计算过程。

```
while True:
    try:
        n=int(input('请输入评委人数：'))
        if n <=2:
            print('评委人数太少，必须多于 2 个人。')
        else:
            break
    except:
        pass

scores=[]
```

```
for i in range(n):
    #这个 while 循环用来保证用户必须输入 0～100 之间的数字
    while True:
        try:
            score = input('请输入第{0}个评委的分数：'.format(i +1))
            #把字符串转换为实数
            score = float(score)
            assert 0 <= score <= 100
            scores.append(score)
            #如果数据合法,跳出 while 循环,继续输入下一个评委的分数
            break
        except:
            print('分数错误')

#计算并删除最高分与最低分
highest = max(scores)
lowest = min(scores)
scores.remove(highest)
scores.remove(lowest)
finalScore = round(sum(scores)/len(scores),2)

formatter = '去掉一个最高分{0}\n去掉一个最低分{1}\n最后得分{2}'
print(formatter.format(highest,lowest,finalScore))
```

第 5 章



代码复用技术(一):函数

在软件开发过程中,经常有很多操作是完全相同或者是非常相似的,仅仅是要处理的数据不同而已,因此经常会在不同的代码位置多次执行相似甚至完全相同的代码块。很显然,从软件设计和代码复用的角度来讲,直接将代码块复制到多个相应的位置然后进行简单修改绝对不是一个好主意。虽然这样可以使得多份复制的代码可以彼此独立地进行修改,但这样不仅增加了代码量,也增加了代码阅读、理解和维护的难度,为代码测试和纠错带来很大的困难。一旦被复制的代码块将来某天被发现存在问题而需要修改,必须要对所有的复制都做同样的正确修改,这在实际中是很难完成的一项任务。更糟糕的情况是,由于代码量的大幅度增加,导致代码之间的关系更加复杂,很可能在修补旧漏洞的同时又引入了新漏洞,维护成本大幅度增加。因此,应尽量减少使用直接复制代码的方式来实现复用。解决这个问题的有效方法是设计函数(function)和类(class)。本章介绍函数的设计与使用,第 6 章介绍面向对象程序设计。

将可能需要反复执行的代码封装为函数,然后在需要该功能的地方调用封装好的函数,不仅可以实现代码的复用,更重要的是可以保证代码的一致性,只需要修改该函数的代码则所有调用位置均得到体现。同时,把大任务拆分成多个函数也是分治法的经典应用,复杂问题简单化,使得软件开发像搭积木一样简单。当然,在实际开发中,需要对函数进行良好的设计和优化才能充分发挥其优势,并不是使用了函数就万事大吉了。在编写函数时,有很多原则需要参考和遵守,例如,不要在同一个函数中执行太多的功能,尽量只让其完成一个高度相关且大小合适的功能,提高模块的内聚性。另外,尽量减少不同函数之间的隐式耦合,例如,减少全局变量的使用,使得函数之间仅通过调用和参数传递来显式体现其相互关系。再就是设计函数时应尽量减少副作用,只实现指定的功能就可以了,不要做多余的事情。最后,在实际项目开发中,往往会把一些通用的函数封装到一个模块中,并把这个通用模块文件放到顶层文件夹中,这样更方便管理。

5.1 函数定义与使用

5.1.1 基本语法

在 Python 中,定义函数的语法如下:

```
def 函数名([参数列表]):
```

```
    '''注释'''
    函数体
```

在 Python 中使用 def 关键字来定义函数,然后是一个空格和函数名称,接下来是一对括号,在括号内是形式参数列表,如果有多个参数则使用逗号分隔开,括号之后是一个冒号和换行,最后是注释和函数体代码。定义函数时在语法上需要注意的问题主要有:①函数形参不需要声明其类型,也不需要指定函数的返回值类型;②即使该函数不需要接收任何参数,也必须保留一对空的括号;③括号后面的冒号必不可少;④函数体相对于 def 关键字必须保持一定的空格缩进。

下面的函数用来计算斐波那契数列中小于参数 n 的所有值:

```
def fib(n):                                              #定义函数,括号里的 n是形参
    '''accept an integer n.
      return the numbers less than n in Fibonacci sequence.'''
    a,b=1,1
    while a<n:
        print(a,end=' ')
        a,b=b,a+b
    print()
```

该函数的调用方式为

```
fib(1000)                                                #调用函数,括号里的 1000 是实参
```

如果代码本身不能提供非常好的可读性,那么最好加上适当的注释来说明。在定义函数时,开头部分的注释并不是必需的,但是如果为函数的定义加上一段注释的话,可以为用户提供友好的提示和使用帮助。例如,可以使用内置函数 help( )来查看函数的使用帮助,并且在调用该函数时输入左侧圆括号之后,立刻就会得到该函数的使用说明,如图 5-1 所示。

```
>>> def fib(n):
	'''accept an integer n.
	  return the numbers less than n in Fibonacci sequence.'''
	a, b = 1, 1
	while a < n:
		print(a, end=' ')
		a, b = b, a+b
	print()


>>> print(fib.__doc__)
accept an integer n.
	  return the numbers less than n in Fibonacci sequence.
>>> help(fib)
Help on function fib in module __main__:

fib(n)
    accept an integer n.
    return the numbers less than n in Fibonacci sequence.

>>> fib(
        (n)
        accept an integer n.
        return the numbers less than n in Fibonacci sequence.
```

图 5-1　使用注释来为用户提示函数使用说明

在 Python 中,定义函数时也不需要声明函数的返回值类型,而是使用 return 语句结束函数执行的同时返回任意类型的值,函数返回值类型与 return 语句返回表达式的类型一

致。不论 return 语句出现在函数的什么位置,一旦得到执行将直接结束函数的执行。如果函数没有 return 语句、有 return 语句但是没有执行到或者执行了不返回任何值的 return 语句,解释器都会认为该函数以 return None 结束,即返回空值。

在编写函数时,应尽量减少副作用,尽量不要修改参数本身,不要修改除返回值以外的其他内容。另外,应充分利用 Python 函数式编程的特点,让自己定义的函数尽量符合纯函数式编程的要求,例如,保证线程安全、可以并行运行等。

5.1.2 函数嵌套定义、可调用对象与修饰器

1. 函数嵌套定义

Python 允许函数的嵌套定义,在函数内部可以再定义另外一个函数。在 2.4 节有一段代码是用来实现可迭代对象与数字四则运算的,当时是使用 lambda 表达式实现的主要功能,如果使用函数嵌套定义的话,代码可以写作:

```
>>>def myMap(iterable,op,value):                    #自定义函数
    if op not in '+-* /':
        return 'Error operator'
    def nested(item):                               #嵌套定义函数
        return eval(repr(item) +op +repr(value))
    return map(nested,iterable)                     #使用在函数内部定义的函数
>>>list(myMap(range(5),'+',5))                      #调用外部函数,不需要关心其内部实现
[5,6,7,8,9]
>>>list(myMap(range(5),'-',5))
[-5,-4,-3,-2,-1]
>>>list(myMap(range(5),'* ',5))
[0,5,10,15,20]
>>>list(myMap(range(5),'/',5))
[0.0,0.2,0.4,0.6,0.8]
```

下面的函数利用函数嵌套定义和递归实现帕斯卡公式 C(n,i) = C(n-1, i) + C(n-1, i-1),进行组合数 C(n,i)的快速求解。

```
def f2(n,i):
    cache2 =dict()

    def f(n,i):
        if n==i or i==0:
            return 1
        elif (n,i) not in cache2:
            cache2[(n,i)] =f(n-1,i) +f(n-1,i-1)
        return cache2[(n,i)]

    return f(n,i)
```

尽管函数嵌套定义使用很方便，也很灵活，但是并不提倡过多使用，因为这样会导致内部的函数反复定义而影响执行效率。

2. 可调用对象

函数属于 Python 可调用对象之一，由于构造方法的存在，类也是可调用的。像 list()、tuple()、dict()、set()这样的工厂函数实际上都是调用了类的构造方法。另外，任何包含__call__()方法的类的对象也是可调用的。下面的代码使用函数的嵌套定义实现了可调用对象的定义：

```
def linear(a,b):
    def result(x):                    #在 Python 中，函数是可以嵌套定义的
        return a*x +b
    return result                     #返回可被调用的函数
```

下面的代码演示了可调用对象类的定义：

```
class linear:
    def __init__(self,a,b):
        self.a,self.b =a,b
    def __call__(self,x):             #这里是关键
        return self.a* x +self.b
```

使用上面的嵌套函数和类这两种方式中任何一个，都可以通过以下的方式来定义一个可调用对象：

```
taxes =linear(0.3,2)
```

然后通过下面的方式来调用该对象：

```
taxes(5)
```

3. 修饰器

修饰器(decorator)是函数嵌套定义的另一个重要应用。修饰器本质上也是一个函数，只不过这个函数接收其他函数作为参数并对其进行一定的改造之后返回新函数。后面第 6 章中的静态方法、类方法、属性等也都是通过修饰器实现的，Python 中还有很多这样的用法。下面的代码演示了修饰器的定义与使用方法，定义其他函数调用之前或之后需要执行的通用代码，可作用于其他任何函数，提高代码复用度。

```
def before(func):                     #定义修饰器
    def wrapper(*args, **kwargs):
        print('Before function called.')
        return func(*args, **kwargs)
    return wrapper

def after(func):                      #定义修饰器
```

```
    def wrapper(*args, **kwargs):
        result = func(* args, * * kwargs)
        print('After function called.')
        return result
    return wrapper

@before
@after
def test():                                        #同时使用两个修饰器改造函数
    print(3)
#调用被修饰的函数
test()
```

和预想的完全一样,上面代码的运行结果为

```
Before function called.
3
After function called.
```

下面的代码通过定义和使用修饰器有效复用了用户名检查功能的代码,关于面向对象编程的知识请参考第 6 章。

```
def check_permission(func):
    def wrapper(*args, **kwargs):
        if kwargs.get('username') != 'admin':
            raise Exception('Sorry. You are not allowed.')
        return func(*args, **kwargs)
    return wrapper

class ReadWriteFile(object):
    #把函数 check_permission 作为装饰器使用
    @check_permission
    def read(self,username,filename):
        return open(filename,'r').read()

    def write(self,username,filename,content):
        open(filename,'a+').write(content)
    #把函数 check_permission 作为普通函数使用
    write = check_permission(write)

t = ReadWriteFile()
print('Originally…')
print(t.read(username = 'admin',filename = r'd:\sample.txt'))
print('Now,try to write to a file…')
t.write(username = 'admin',filename = r'd:\sample.txt',content = '\nhello world')
print('After calling to write…')
```

```
print(t.read(username='admin',filename=r'd:\sample.txt'))
```

下面的代码使用修饰器对函数进行改写,使用字典存储临时结果,避免了重复计算,从而提高了代码的执行效率,并且有效避免了因为递归深度过深而导致的内存不足。然后使用修饰器定义了函数 f3(),使用帕斯卡公式快速求解组合数,与本节开始提到的嵌套函数的例子具有相同的功能,但是效率更高一些。

```
from functools import wraps

#定义修饰器
def cachedFunc(func):
    #使用字典存储中间结果
    cache=dict()
    #对目标函数进行改写
    @wraps(func)
    def newFunc(*args):
        if args not in cache:
            cache[args]=func(*args)
        return cache[args]
    #返回修改过的新函数
    return newFunc

#使用修饰器
@cachedFunc
def f3(n,i):
    if n==i or i==0:
        return 1
    return f3(n-1,i) +f3(n-1,i-1)
```

同样的修饰器也适用于斐波那契数列的递归求解函数。

```
@cachedFunc
def fib1(i):
    if i<2:
        return 1
    return fib1(i-1) +fib1(i-2)
```

为了实现上述功能,Python 标准库 functools 提供了一个缓存修饰器 lru_cache,可以大幅度提高代码的性能,其中的参数 maxsize 用来限制缓存的条目数量,当缓存的条目数达到最大时会自动删除最近最少使用的条目。例如下面的代码,比使用上面的修饰器效果要好很多。

```
import functools
@functools.lru_cache(maxsize=64)
def f4(n,i):
    if n==i or i==0:
```



```
        return 1
    return f4(n-1,i) +f4(n-1,i-1)
```

5.1.3　函数对象成员的动态性

正如第 1 章所说，Python 是一种高级动态编程语言，变量类型是随时可以改变的。Python 中的函数和自定义对象的成员也是可以随时发生改变的，可以为函数和自定义对象动态增加新成员。本节介绍函数的动态性，第 6 章再介绍自定义对象的动态性。

```
>>>def func():
    print(func.x)                            #查看函数 func 的成员 x
>>>func()                                    #现在函数 func 还没有成员 x,出错
AttributeError: 'function' object has no attribute 'x'
>>>func.x=3                                  #动态为函数增加新成员
>>>func()
3
>>>func.x                                    #在外部也可以直接访问函数的成员
3
>>>del func.x                                #删除函数成员
>>>func()                                    #删除之后不可访问
AttributeError: 'function' object has no attribute 'x'
```

5.1.4　函数递归调用

函数的递归调用是函数调用的一种特殊情况，函数调用自己，自己再调用自己，自己再调用自己……，当某个条件得到满足的时候就不再调用了，然后再一层一层地返回直到该函数的第一次调用，如图 5-2 所示。

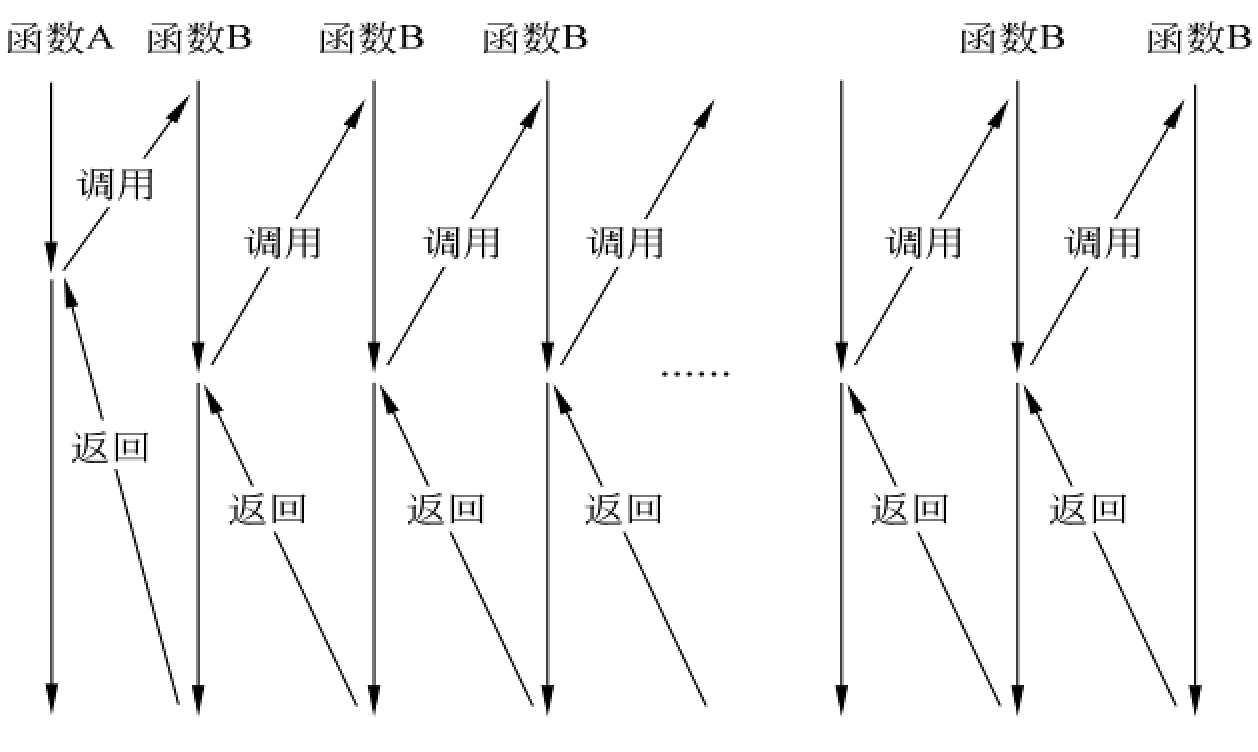


图 5-2　函数递归调用示意图

函数递归通常用来把一个大型的复杂问题层层转化为一个与原来问题本质相同但规模很小、很容易解决或描述的问题，只需要很少的代码就可以描述解决问题过程中需要的

大量重复计算。下面的代码使用递归计算列表中所有元素之和,尽管在 Python 没有这样做的必要。

```
def recursiveSum(lst):
    if len(lst) ==1:
        return lst[0]
    return lst[0] +recursiveSum(lst[1:])
```

而下面的代码则使用递归实现了整数的因数分解,函数执行结束后,fac 中包含了整数 num 因数分解的结果。

```
from random import randint

def factors(num,fac=[]):
    #每次都从 2 开始查找因数
    for i in range(2,int(num**0.5) +1):
        #找到一个因数
        if num% i ==0:
            fac.append(i)
            #对商继续分解,重复这个过程
            factors(num//i,fac)
            #注意,这个 break 非常重要
            break
    else:
        #不可分解了,自身也是个因数
        fac.append(num)

facs =[]
n =randint(2,10 **8)
factors(n,facs)
result ='* '.join(map(str,facs))
if n==eval(result):
    print(n,'=' +result)
```

最后,从图 5-2 中可以看出,每次调用函数时必须要记住离开的位置才能保证函数运行结束以后回到正确的位置,这个过程称为保存现场,这需要一定的栈空间。另外,调用一个函数时会为该函数分配一个栈帧,用来存放普通参数和函数内部局部变量的值,这个栈帧会在函数调用结束后自动释放。而在函数递归调用的情况中,一个函数执行尚未结束就又调用了自己,原来的栈帧还没释放又分配了新栈帧,会占用大量的栈空间。因此,递归深度如果太大的话,可能会导致栈空间不足进而导致程序崩溃。

5.2 函数参数

函数定义时括号内是使用逗号分隔开的形参列表(parameters),函数可以有多个参数,也可以没有参数,但定义和调用时一对括号必须要有,表示这是一个函数并且不接收

参数。调用函数时向其传递实参(arguments)，根据不同的参数类型，将实参的值或引用传递给形参。定义函数时不需要声明参数类型，解释器会根据实参的类型自动推断形参类型，在一定程度上类似于函数重载和泛型函数的功能。

一般来说，在函数内部直接修改形参的值不会影响实参。例如：

```
>>>def addOne(a):
    a +=1                                    #这条语句会得到一个新的变量 a
>>>a =3
>>>addOne(a)
>>>a                                         #实参的值没有受到影响
3
```

从运行结果可以看出，在函数内部修改了形参 a 的值，但是当函数运行结束以后，实参 a 的值并没有被修改。然而，列表、字典、集合这样的可变序列类型作为函数参数时，如果在函数内部通过列表、字典或集合对象自身的方法修改参数中的元素时，同样的作用也会体现到实参上。

```
>>>def modify(v):                            #修改列表元素值
    v[0] =v[0] +1
>>>a =[2]
>>>modify(a)
>>>a
[3]
>>>def modify(v,item):                       #为列表增加元素
    v.append(item)
>>>a =[2]
>>>modify(a,3)
>>>a
[2,3]
>>>def modify(d):                            #修改字典元素值或为字典增加元素
    d['age'] =38
>>>a ={'name':'Dong','age':37,'sex':'Male'}
>>>modify(a)
>>>a
{'age': 38,'name': 'Dong','sex': 'Male'}
>>>def modify(s,v):                          #为集合添加元素
    s.add(v)
>>>s ={1,2,3}
>>>modify(s,4)
>>>s
{1,2,3,4}
```

也就是说，如果传递给函数的是列表、字典、集合或其他自定义的可变序列，并且在函数内部使用下标或序列自身支持的方式为可变序列增加、删除元素或修改元素值时，修改

后的结果是可以反映到函数之外的,即实参也得到了相应的修改。

第 2 章和第 3 章曾经多次提到,Python 采用的是基于值的自动内存管理模式,变量并不直接存储值,而是存储值的引用。从这个角度来讲,在 Python 中调用函数时,实参到形参都是传递的引用。也就是说,Python 函数不存在传值调用。

5.2.1 位置参数

位置参数(positional arguments)是比较常用的形式,调用函数时实参和形参的顺序必须严格一致,并且实参和形参的数量必须相同。

```
>>>def demo(a,b,c):                          #所有形参都是位置参数
    print(a,b,c)
>>>demo(3,4,5)
3 4 5
>>>demo(3,5,4)
3 5 4
>>>demo(1,2,3,4)                             #实参与形参的数量必须相同
TypeError: demo() takes 3 positional arguments but 4 were given
```

5.2.2 默认值参数

在定义函数时,Python 支持默认值参数,在定义函数时可以为形参设置默认值。在调用带有默认值参数的函数时,可以不用为设置了默认值的形参进行传值,此时函数将会直接使用函数定义时设置的默认值,当然也可以通过显式赋值来替换其默认值。也就是说,**在调用函数时是否为默认值参数传递实参是可选的**,具有较大的灵活性,在一定程度上类似于函数重载的功能,同时还能在为函数增加新的参数和功能时通过为新参数设置默认值来保证向后兼容而不影响老用户的使用。需要注意的是,在定义带有默认值参数的函数时,**任何一个默认值参数右边都不能再出现没有默认值的普通位置参数**,否则会提示语法错误。带有默认值参数的函数定义语法如下:

```
def 函数名(…,形参名=默认值):
    函数体
```

可以使用“函数名.__defaults__”随时查看函数所有默认值参数的当前值,其返回值为一个元组,其中的元素依次表示每个默认值参数的当前值。

```
>>>def say(message,times=1):
    print((message+' ') * times)
>>>say.__defaults__
(1,)
```

调用该函数时,如果只为第一个参数传递实参,则第二个参数使用默认值 1,如果为第二个参数传递实参,则不再使用默认值 1,而是使用调用者显式传递的值。




```
>>>say('hello')
hello
>>>say('hello',3)
hello hello hello
```

多次调用函数并且不为默认值参数传递值时,默认值参数只在定义时进行一次解释和初始化,对于列表、字典这样可变类型的默认值参数,这一点可能会导致很严重的逻辑错误,而这种错误或许会耗费大量精力来定位和纠正。

```
>>>def demo(newitem,old_list=[]):
    old_list.append(newitem)
    return old_list

>>>print(demo('5',[1,2,3,4]))
[1,2,3,4,'5']
>>>print(demo('aaa',['a','b']))
['a','b','aaa']
>>>print(demo('a'))
['a']
>>>print(demo('b'))                              #注意这里的输出结果
['a','b']
```

上面的函数使用列表作为默认参数,由于其可记忆性,连续多次调用该函数而不给该参数传值时,再次调用时将保留上一次调用的结果。一般来说,要避免使用列表、字典、集合或其他可变序列作为函数参数默认值,对于上面的函数,更建议使用下面的写法。

```
def demo(newitem,old_list=None):
    if old_list is None:
        old_list=[]
    old_list.append(newitem)
    return old_list
```

另外一个需要注意的问题是,如果在定义函数时某个参数的默认值为另一个变量的值,那么参数的默认值只依赖于函数定义时该变量的值,或者说函数的默认值参数是在函数定义时确定值的,所以只会被初始化一次。例如:

```
>>>i=3
>>>def f(n=i):                                   #参数 n 的值仅取决于 i 的当前值
    print(n)
>>>f()
3
>>>i=5                                           #函数定义后修改 i 的值不影响参数 n 的默认值
>>>f()
3
>>>i=7
>>>f()
```

```
3
>>>def f(n=i):                             #重新定义函数
    print(n)
>>>f()
7
```

5.2.3 关键参数

关键参数主要指调用函数时的参数传递方式，与函数定义无关。通过关键参数可以按参数名字传递值，明确指定哪个值传递给哪个参数，实参顺序可以和形参顺序不一致，但不影响参数值的传递结果，避免了用户需要牢记参数位置和顺序的麻烦，使得函数的调用和参数传递更加灵活方便。

```
>>>def demo(a,b,c=5):
    print(a,b,c)
>>>demo(3,7)                               #按位置传递参数
3 7 5
>>>demo(c=8,a=9,b=0)                       #关键参数
9 0 8
```

5.2.4 可变长度参数

可变长度参数在定义函数时主要有两种形式：*parameter 和 **parameter，前者用来接收任意多个实参并将其放在一个元组中，后者接收类似于关键参数一样显式赋值形式的多个实参并将其放入字典中。

下面的代码演示了第一种形式可变长度参数的用法，无论调用该函数时传递了多少实参，一律将其放入元组中：

```
>>>def demo(*p):
  print(p)
>>>demo(1,2,3)
(1,2,3)
>>>demo(1,2,3,4,5,6,7)
(1,2,3,4,5,6,7)
```

下面的代码演示了第二种形式可变长度参数的用法，即在调用该函数时自动将接收的参数转换为字典：

```
>>>def demo(**p):
  for item in p.items():
    print(item)
>>>demo(x=1,y=2,z=3)
('y',2)
('x',1)
```

```
('z',3)
```

在函数定义中增加一些外围的检查代码，可以防止用户调用函数时传入无效参数。

```
>>>def demo(a,b, **kwargs):
    for para in kwargs.keys():
        #只允许传入 x、y、z 这样的关键参数
        if para not in ('x','y','z'):
            raise Exception('{0} is not a valid parameter'.format(para))
    print(a+b+sum(kwargs.values()))

>>>demo(1,2)                              #可以不给 kwargs 传值,默认为空字典
3
>>>demo(1,2,x=3)
6
>>>demo(1,2, xx=3)                        #不接收 xx 这样的参数名
Exception: xx is not a valid parameter
```

Python 定义函数时可以同时使用位置参数、关键参数、默认值参数和可变长度参数，但是除非真的很必要，否则不要这样做，因为这会使得代码非常混乱而严重降低可读性，并导致程序查错非常困难。另外，一般而言，一个函数如果可以接收很多不同类型参数的话，很可能是函数设计得不好，例如函数功能过多，需要进行必要的拆分和重新设计，以满足模块高内聚的要求。

5.2.5　强制函数的某些参数必须以关键参数形式进行传值

在函数定义时，位于 * parameter 或单独一个星号 * 之后的所有参数都只能以关键参数的形式进行传值，不接收其他任何形式的传值。

```
>>>def demo(a,b, * ,c):                   #参数 c 必须以关键参数进行传值
    print(a+b+c)
>>>demo(1,2,c=3)                          #正确
6
>>>demo(1,2,3)                            #错误,引发异常
TypeError: demo() takes 2 positional arguments but 3 were given
>>>def demo(a,b, *p,c):                   #参数 c 必须以关键参数进行传值
    print(a+b+c+sum(p))
>>>demo(1,2,3,4,c=5)                      #正确
15
>>>demo(1,2,3,4,5)                        #错误,引发异常
TypeError: demo() missing 1 required keyword-only argument: 'c'
```

如果需要强制函数的所有参数都必须以关键参数形式进行传值，可以在定义函数时把单独一个星号 * 作为函数第一个参数。例如：

```
>>>def demo(* ,a,b):
```

```
    print(a,b)
>>>demo(a=1,b=2)
1 2
>>>demo(1,2)
TypeError: demo() takes 0 positional arguments but 2 were given
```

也可以使用修饰器实现同样的功能,下面的代码首先定义了一个修饰器对函数的关键参数和位置参数进行检查,如果发现有参数没有以关键参数形式传值则抛出异常。

```
def mustBeKeywords(func):
    import inspect
    #获取位置参数和默认值参数列表
    positions=inspect.getargspec(func).args
    def wrapper(*args, **kwargs):
        for pos in positions:
            if pos not in kwargs:
                raise Exception(pos+' must be keyword parameter')
            return func(*args, **kwargs)
    return wrapper

@mustBeKeywords
def demo(a,b,c):
    print(a,b,c)
```

5.2.6 强制函数的所有参数必须以位置参数形式进行传值

在 2.4 节曾经提到,内置函数 sum()强制要求参数必须按位置进行传值而不允许使用关键参数的形式,使用 help()查看帮助文档的话会发现 sum()函数的最后一个参数是斜线。那是 Argument Clinic 的设计,与底层 C 语言实现有关,在 Python 中定义函数时不允许使用斜线作为函数参数。尽管不允许使用斜线作为参数,但是通过修饰器也可以实现同样的效果。例如下面的代码,首先定义了一个修饰器对函数的位置参数和关键参数进行检查,如果发现有关键参数与位置参数同名则抛出异常。

```
def onlyPositions(func):
    import inspect
    #获取函数 func 的位置参数列表
    positions=inspect.getargspec(func).args
    def wrapper(*args, **kwargs):
        #检查关键参数列表
        for para in kwargs:
            if para in positions:
                raise Exception(para+' can not be keyword parameter')
            return func(*args, **kwargs)
    return wrapper
```

```
@onlyPositions
def demo(a,b,c):
    print(a,b,c)
```

5.2.7　传递参数时的序列解包

与可变长度的参数相反,这里的序列解包是指实参,同样也有 * 和 **两种形式。调用含有多个位置参数(positional arguments)的函数时,可以使用 Python 列表、元组、集合、字典以及其他可迭代对象作为实参,并在实参名称前加一个星号,Python 解释器将自动进行解包,然后把序列中的值分别传递给多个单变量形参。

```
>>>def demo(a,b,c):                         #可以接收多个位置参数的函数
    print(a+b+c)
>>>seq=[1,2,3]
>>>demo(*seq)                               #对列表进行解包
6
>>>tup=(1,2,3)
>>>demo(*tup)                               #对元组进行解包
6
>>>dic={1:'a',2:'b',3:'c'}
>>>demo(*dic)                               #对字典的键进行解包
6
>>>demo(*dic.values())                      #对字典的值进行解包
abc
>>>Set={1,2,3}
>>>demo(*Set)                               #对集合进行解包
6
```

如果实参是个字典,可以使用两个星号 **对其进行解包,会把字典转换成类似于关键参数的形式进行参数传递。对于这种形式的序列解包,要求实参字典中的所有键都必须是函数的形参名称,或者与函数中两个星号的可变长度参数相对应。

```
>>>p={'a':1,'b':2,'c':3}                    #要解包的字典
>>>def f(a,b,c=5):                          #带有位置参数和默认值参数的函数
    print(a,b,c)
>>>f(**p)
1 2 3
>>>def f(a=3,b=4,c=5):                      #带有多个默认值参数的函数
    print(a,b,c)
>>>f(**p)                                   #对字典元素进行解包
1 2 3
>>>def demo(**p):                           #接收字典形式可变长度参数的函数
    for item in p.items():
        print(item)
```

```
>>>p={'x':1,'y':2,'z':3}
>>>demo(**p)                              #对字典元素进行解包
('y',2)
('z',3)
('x',1)
```

如果一个函数需要以多种形式来接收参数,定义时一般把位置参数放在最前面,然后是默认值参数,接下来是一个星号的可变长度参数,最后是两个星号的可变长度参数;调用函数时,一般也按照这个顺序进行参数传递。调用函数时如果对实参使用一个星号 * 进行序列解包,那么这些解包后的实参将会被当作普通位置参数对待,并且会在关键参数和使用两个星号 ** 进行序列解包的参数之前进行处理。

```
>>>def demo(a,b,c):                       #定义函数
    print(a,b,c)
>>>demo(*(1,2,3))                         #调用,序列解包
1 2 3
>>>demo(1, *(2,3))                        #位置参数和序列解包同时使用
1 2 3
>>>demo(1, *(2,),3)
1 2 3
>>>demo(a=1, *(2,3))                      #一个星号的序列解包相当于位置参数
                                          #优先处理,引发异常
TypeError: demo() got multiple values for argument 'a'
>>>demo(b=1, *(2,3))                      #重复给 b 赋值,引发异常
TypeError: demo() got multiple values for argument 'b'
>>>demo(c=1, *(2,3))                      #一个星号的序列解包相当于位置参数
                                          #优先处理
2 3 1
>>>demo(**{'a':1,'b':2}, * (3,))          #序列解包不能在关键参数解包之后
SyntaxError: iterable argument unpacking follows keyword argument unpacking
>>>demo(*(3,), **{'a':1,'b':2})           #一个星号的序列解包相当于位置参数
                                          #优先处理,引发异常
TypeError: demo() got multiple values for argument 'a'
>>>demo(*(3,), **{'c':1,'b':2})
3 2 1
```

5.2.8 标注函数参数与返回值类型

虽然 Python 是一种强类型语言,但它并不允许直接声明变量的类型。根据赋值语句来自动推断变量类型,根据函数调用时传递的实参来自动推断形参类型,这不仅体现了变量的动态性,更重要的是提高了函数的适用范围,有利于代码复用。但在实际开发中,有时候可能需要限制函数所能接收的实参类型,虽然 Python 允许使用一种标注函数参数和返回值类型的形式,但实际上并不起作用。如果确实需要对函数参数和返回值类型进行

严格的限制,可以考虑使用断言语句 assert、异常处理结构或 if…else 语句再结合 type( )、isinstance( )之类的类型检查函数来确保参数类型。

下面的函数要求参数必须为整数,如果接收到任何其他类型的参数都会抛出异常。如果把其中的几个 assert 语句删除或者注释掉的话就会发现,所谓标注函数参数和返回值类型的形式并不起什么作用,只是看上去比较清晰而已,真正起作用的是其中的 assert 语句。

```
>>>def test(x:int,y:int) ->int:
    '''x and y must be integers,return an integer x+y'''
    assert isinstance(x,int),'x must be integer'
    assert isinstance(y,int),'y must be integer'
    z=x+y
    assert isinstance(z,int),'must return an integer'
    return z
>>>test(1,2)
3
>>>test(2,3.0)                                   #参数类型不符合要求,抛出异常
AssertionError: y must be integer
```

5.3　变量作用域

5.3.1　全局变量与局部变量

变量起作用的代码范围称为变量的作用域,不同作用域内同名变量之间互不影响,就像不同文件夹中的同名文件之间互不影响一样。在函数外部和在函数内部定义的变量,其作用域是不同的,函数内部定义的变量一般为局部变量,在函数外部定义的变量为全局变量。不管是局部变量还是全局变量,其作用域都是从定义的位置开始的,在此之前无法访问。

在函数内定义的局部变量只在该函数内可见,当函数运行结束后,在其内部定义的所有局部变量将被自动删除而不可访问。在函数内部使用 global 定义的全局变量当函数结束以后仍然存在并且可以访问。

如果想要在函数内部修改一个定义在函数外的变量值,必须要使用 global 明确声明,否则会自动创建新的局部变量。在函数内部通过 global 关键字来声明或定义全局变量,这分两种情况。

(1) 一个变量已在函数外定义,如果在函数内需要修改这个变量的值,并将修改的结果反映到函数之外,可以在函数内用关键字 global 明确声明要使用已定义的同名全局变量。

(2) 在函数内部直接使用 global 关键字将一个变量声明为全局变量,如果在函数外没有定义该全局变量,在调用这个函数之后,会创建新的全局变量。



或者说，也可以这么理解：①在函数内如果只引用某个变量的值而没有为其赋新值，该变量为(隐式的)全局变量；②如果在函数内某条代码有为变量赋值的操作，该变量就被认为是(隐式的)局部变量，除非在函数内赋值操作之前显式地用关键字 global 进行了声明。

下面的代码演示了局部变量和全局变量的用法。

```
>>>def demo():
    global x                          #声明或创建全局变量,必须在使用 x之前执行
    x=3                               #修改全局变量的值
    y=4                               #局部变量
    print(x,y)
>>>x=5                                #在函数外部定义了全局变量 x
>>>demo()                             #本次调用修改了全局变量 x的值
3 4
>>>x
3
>>>y                                  #局部变量在函数运行结束之后自动删除,不再存在
NameError: name 'y' is not defined
>>>del x                              #删除了全局变量 x
>>>x
NameError: name 'x' is not defined
>>>demo()                             #本次调用创建了全局变量
3 4
>>>x
3
```

如果在某个作用域内有为变量赋值的操作，那么该变量将被认为是该作用域内的局部变量，这一点一定要引起注意。

```
>>>x=10                               #全局变量
>>>def demo():
    print(x)                          #这条语句会引发异常,因为 x变量现在还不存在
    x=x+1                             #赋值语句,x将被认为是该作用域内的局部变量
    print(x)

>>>demo()
UnboundLocalError: local variable 'x' referenced before assignment
```

如果局部变量与全局变量具有相同的名字，那么该局部变量会在自己的作用域内暂时隐藏同名的全局变量。

```
>>>def demo():
    x=3                               #创建了局部变量,并自动隐藏了同名的全局变量
    print(x)
```

```
>>>x=5                                        #创建全局变量
>>>x
5
>>>demo()
3
>>>x                                          #函数调用结束后,不影响全局变量 x的值
5
```

如果需要在不同模块之间共享全局变量的话,可以编写一个专门的模块来实现这一目的。例如,假设在模块 A. py 中有如下变量定义:

```
global_variable=0
```

而在模块 B. py 中使用以下语句修改该全局变量的值:

```
import A
A.global_variable=1
```

在模块 C. py 中使用以下语句来访问全局变量的值:

```
import A
print(A.global_variable)
```

一般而言,局部变量的引用速度比全局变量要快一些,应优先考虑使用。并且应该尽量避免过多使用全局变量,因为全局变量会增加不同函数之间的隐式耦合度,降低代码可读性,并使得代码测试和纠错变得很困难。

局部变量的空间是在栈上分配的,而栈空间是由操作系统维护的,每当调用一个函数时,操作系统会为其分配一个栈帧,函数调用结束后立刻释放这个栈帧。因此,函数调用结束后,该函数内部所有的局部变量都不再存在。

内置函数 globals( ) 和 locals( ) 分别返回包含当前作用域内所有全局变量和局部变量的名称和值的字典。

```
>>>a=(1,2,3,4,5)                              #全局变量
>>>b='Hello world.'                           #全局变量
>>>def demo():
    a=3                                       #局部变量
    b=[1,2,3]                                 #局部变量
    print('locals:',locals())
    print('globals:',globals())
>>>demo()
locals: {'a': 3,'b': [1,2,3]}
globals: {'a': (1,2,3,4,5),'b': 'Hello world.','__builtins__': <module '__builtin__' (built-in)
>,'demo': <function demo at 0x013907F0 >,'__package__': None,'__name__': '__main__','__doc__
': None}
```

5.3.2 nonlocal 变量

除了局部变量和全局变量,Python 还支持使用 nonlocal 关键字定义一种介于两者之间的变量。关键字 nonlocal 声明的变量会引用距离最近的非全局作用域的变量,要求声明的变量已经存在,关键字 nonlocal 不会创建新变量。

```
def scope_test():
    def do_local():
        spam="我是局部变量"

    def do_nonlocal():
        nonlocal spam                    #这时要求 spam 必须是已存在的变量
        spam="我不是局部变量,也不是全局变量"

    def do_global():
        global spam                      #如果全局作用域内没有 spam,就自动新建一个
        spam="我是全局变量"

    spam="原来的值"
    do_local()
    print("局部变量赋值后:",spam)
    do_nonlocal()
    print("nonlocal 变量赋值后:",spam)
    do_global()
    print("全局变量赋值后:",spam)

scope_test()
print("全局变量:",spam)
```

上面的代码运行结果为

```
局部变量赋值后:原来的值
nonlocal 变量赋值后:我不是局部变量,也不是全局变量
全局变量赋值后:我不是局部变量,也不是全局变量
全局变量:我是全局变量
```

5.4 lambda 表达式

lambda 表达式常用来声明匿名函数,即没有函数名字的临时使用的小函数,常用在临时需要一个类似于函数的功能但又不想定义函数的场合,例如,内置函数 sorted( )和列表方法 sort( )的 key 参数,内置函数 map( )和 filter( )的第一个参数,等等。lambda 表达式只可以包含一个表达式,不允许包含其他复杂的语句,但在表达式中可以调用其他函数,该表达式的计算结果相当于函数的返回值。下面的代码演示了不同情况下 lambda 表



达式的应用。

```
>>>f=lambda x,y,z: x+y+z                          #也可以给 lambda 表达式起个名字
>>>print(f(1,2,3))                                #把 lambda 表达式当作函数使用
6
>>>g=lambda x,y=2,z=3: x+y+z                      #支持默认值参数
>>>print(g(1))
6
>>>print(g(2,z=4,y=5))                            #调用时使用关键参数
11
>>>L=[(lambda x: x**2),(lambda x: x**3),(lambda x: x**4)]
>>>print(L[0](2),L[1](2),L[2](2))
4 8 16
>>>D={'f1':(lambda: 2+3),'f2':(lambda: 2*3),'f3':(lambda: 2**3)}
>>>print(D['f1'](),D['f2'](),D['f3']())
5 6 8
>>>L=[1,2,3,4,5]
>>>list(map(lambda x: x+10,L))                    #lambda 表达式作为函数参数
[11,12,13,14,15]
>>>def demo(n):
    return n*n
>>>demo(5)
25
>>>a_list=[1,2,3,4,5]
>>>list(map(lambda x: demo(x),a_list))            #在 lambda 表达式中可以调用函数
[1,4,9,16,25]
>>>data=list(range(20))
>>>import random
>>>random.shuffle(data)
>>>data
[4,3,11,13,12,15,9,2,10,6,19,18,14,8,0,7,5,17,1,16]
>>>data.sort(key=lambda x: x)                     #用在列表的 sort()方法中,作为函数参数
>>>data
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19]
>>>data.sort(key=lambda x: len(str(x)))           #使用 lambda 表达式指定排序规则
>>>data
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19]
>>>data.sort(key=lambda x: len(str(x)),reverse=True)
>>>data
[10,11,12,13,14,15,16,17,18,19,0,1,2,3,4,5,6,7,8,9]
```

在使用 lambda 表达式时,要注意变量作用域可能会带来的问题,例如,下面的代码中变量 x 是在外部作用域中定义的,对 lambda 表达式而言不是局部变量,从而导致出现了错误。

```
>>>r=[]
>>>for x in range(10):
    r.append(lambda: x**2)
>>>r[1]()                                          #请自行验证 r[0]()、r[2]()、r[3]()和其他几个
81
>>>x=3
>>>r[1]()
9
```

而修改为下面的代码,则可以得到正确的结果。

```
>>>r=[]
>>>for x in range(10):
    r.append(lambda n=x: n**2)
>>>r[0]()
0
>>>r[1]()
1
>>>r[5]()                                          #请自行验证其他几个
25
```

或许下面这个例子更能说明问题,这里的 lambda 表达式相当于只有一条 return i 语句的小函数,调用时真正的返回值取决于全局变量 i 的当前值。

```
>>>f=lambda :i
>>>i=3
>>>f()
3
>>>i=5
>>>f()
5
```

作为一种花哨的技巧,可以定义嵌套的 lambda 表达式,再结合内置函数 map( )、filter( )以及其他函数,可以实现一些复杂的计算,尽管并不建议这样用。例如,下面的代码输出 100 之内的所有素数,为了看得更清楚,作者拆成了几行并使用缩进表示了各部分的关系,尽管这样,相信大家还是觉得很难理解。

```
print(list(filter(None,
                  map(lambda y:y* reduce(lambda x,y:x* y!=0,
                                         map(lambda x,y=y:y%x,
                                             range(2,int(y**0.5 +1))),
                                         1),range(2,1000)))))
```

下面的代码输出斐波那契数列的前 10 个数字,相当于函数的递归调用,和上一段代码一样,并不是很容易理解。

```
print(list(map(lambda x,f=lambda x,f:(f(x-1,f) +f(x-2,f)) if x>1 else 1:f(x,f),range(10))))
```

最后应注意，虽然使用 lambda 表达式可以很方便灵活地定义一些小函数，但是，如果仅仅是需要一个简单的运算，那么应该尽量使用标准库 operator 中提供的函数，避免自己定义 lambda 表达式，operator 中的函数执行效率更高一些。

5.5　生成器函数设计要点

包含 yield 语句的函数可以用来创建生成器对象，这样的函数也称生成器函数。yield 语句与 return 语句的作用相似，都是用来从函数中返回值。return 语句一旦执行会立刻结束函数的运行，而每次执行到 yield 语句并返回一个值之后会暂停或挂起后面代码的执行，下次通过生成器对象的__next__()方法、内置函数 next()、for 循环遍历生成器对象元素或其他方式显式“索要”数据时恢复执行。**生成器具有惰性求值的特点**，适合大数据处理。下面的代码演示了如何使用生成器来生成斐波那契数列：

```
>>>def f():
    a,b=1,1                                 #序列解包,同时为多个元素赋值
    while True:
        yield a                             #暂停执行,需要时再产生一个新元素
        a,b=b,a+b                           #序列解包,继续生成新元素
>>>a=f()                                    #创建生成器对象
>>>for i in range(10):                      #斐波那契数列中前 10 个元素
    print(a.__next__(),end=' ')
1 1 2 3 5 8 13 21 34 55
>>>for i in f():                            #斐波那契数列中第一个大于 100 的元素
    if i>100:
        print(i,end=' ')
        break
144
>>>a=f()                                    #创建生成器对象
>>>next(a)                                  #使用内置函数 next()获取生成器对象中的元素
1
>>>next(a)                                  #每次索取新元素时,由 yield 语句生成
1
>>>a.__next__()                             #也可以调用生成器对象的__next__()方法
2
>>>a.__next__()
3
>>>def f():
    yield from 'abcdefg'                    #使用 yield 表达式创建生成器

>>>x=f()
>>>next(x)
'a'
>>>next(x)
'b'
```

```
>>>for item in x:                                    #输出 x 中的剩余元素
    print(item,end=' ')

c d e f g
>>>def gen():
    yield 1
    yield 2
    yield 3

>>>x,y,z =gen()                                      #生成器对象支持序列解包
```

生成器对象还支持使用 send( ) 方法传入新值,从而改变后续生成的数据,这时要对 yield 表达式稍微改写一下。

```
>>>def gen(start,end):
    i =start
    while i <end:
        v =(yield i)
        if v:
            i =v
        else:
            i +=1

>>>g =gen(1,101)
>>>next(g)
1
>>>g.__next__()
2
>>>g.send(9)                                         #传入新值,改变后续生成的数据
9
>>>next(g)
10
```

Python 标准库 itertools 提供了一个 count(start, step) 函数,用来连续不断地生成无穷个数,这些数中的第一个数是 start(默认为 0),相邻两个数的差是 step(默认为 1)。下面的代码使用生成器模拟了标准库 itertools 中的 count( ) 函数。

```
>>>def count(start,step):
    num =start
    while True:                                      #无穷循环
        yield num                                    #返回一个数,暂停执行,等待下一次索要数据
        num +=step
>>>x =count(3,5)
>>>for i in range(10):
    print(next(x),end=' ')
3 8 13 18 23 28 33 38 43 48
>>>for i in range(10):
```



```
    print(next(x),end=' ')
53 58 63 68 73 78 83 88 93 98
```

Python 标准库 inspect 中的 isgeneratorfunction( )函数可以用来判断一个函数是否为生成器函数。

```
>>>def test(a,b):                              #普通函数
    return a+b
>>>import inspect
>>>inspect.isgeneratorfunction(test)
False
>>>def test():                                 #生成器函数
    yield from (1,2,3)
    for i in range(4,6):
        yield i
>>>inspect.isgeneratorfunction(test)
True
>>>t=test()
>>>for item in t:                              #遍历生成器对象中的元素
    print(item,end=' ')
1 2 3 4 5
```

5.6　偏函数与函数柯里化

偏函数(partial function)和函数柯里化(function currying)是函数式编程中常用的技术。有时候人们在复用已有函数时可能需要固定其中的部分参数,这除了可以通过默认值参数来实现之外,还可以使用偏函数。

例如,下面的代码创建了内置函数 print( )的偏函数并进行调用。

```
>>>from functools import partial
>>>func=partial(print,'Hello world.',flush=True)
>>>func()
Hello world.
```

再例如,有个函数用来实现 3 个数字相加:

```
def add3(a,b,c):
    return a+b+c
```

如果现在需要一个类似的函数,与上面的函数 add3( )的区别仅在于参数 b 固定为一个数字(例如 666),这时就可以使用偏函数的技术来复用上面的函数。

```
def add2(a,c):
    return add3(a,666,c)
print(add2(1,1))
```

或者使用标准库 functools 提供的 partial( )方法创建指定函数的偏函数。

```
from functools import partial
add2 =partial(add3,b =666)
print(add2(a =1,c =1))
```

再例如,下面的代码创建了 int( )的偏函数,并固定其参数 base 为 2。

```
from functools import partial
int2 =partial(int,base =2)
print(int2('111'))
```

函数柯里化除了可以实现偏函数类似的功能之外,还可以利用单参数函数来实现多参数函数,这要归功于 Python 对函数嵌套定义和 lambda 表达式的支持。5.1.2 节介绍的嵌套函数其实就说明了函数柯里化的用法,这里再举一个例子。

```
def func(a):
    return lambda b: a +b
print(func(3)(5))
```

或者

```
def func(a):
    def funcNested(b):
        return a +b
    return funcNested
print(func(3)(5))
```

当然,也可以多级嵌套定义函数实现更多参数的需求。

```
def func(a):
    def funcNested(b):
        def funcNestedNested(c):
            return a +b +c
        return funcNestedNested
    return funcNested
print(func(3)(5)(8))
```

5.7 单分发器与泛型函数

这里的泛型函数(generic function)是指由一组为不同类型参数执行相似操作的函数组成的函数,具体调用哪一个函数的实现取决于分发算法和参数类型。Python 单分发器是实现泛型函数的一种形式,由一个单一参数来决定选择和调用哪个函数。下面的代码演示了单分发器泛型函数的有关用法:

```
from functools import singledispatch

@singledispatch
def fun(arg,verbose =False):
```

```
    '''如果没有合适的函数,就调用这个函数'''
    if verbose:
        print('Let me just say,',end=' ')
    print(arg)

#把 register 当作修饰器使用,为不同类型的参数分别创建不同的实现
#使用下画线表示不关心函数的具体名字
@fun.register(int)
def _(arg,verbose=False):
    '''如果第一个参数的类型是 int,就调用这个函数'''
    if verbose:
        print('Strength in numbers,',end=' ')
    print(arg)

#也可以为函数起个名字
@fun.register(float)
def fun_num(arg,verbose=False):
    '''如果第一个参数的类型是 float,就调用这个函数'''
    if verbose:
        print('Half of your number is:',end=' ')
    print(arg/2)

@fun.register(list)
@fun.register(tuple)
def _(arg,verbose=False):
    '''如果第一个参数的类型是 list 或 tuple,就调用这个函数'''
    if verbose:
        print('Enumerate this:')
    for i,v in enumerate(arg):
        print(i,v)

#自定义类
class Scores:
    def __init__(self, *score):
        self.score=list(score)
#为自定义类型创建泛型函数
@fun.register(Scores)
def _(arg,verbose=False):
    if verbose:
        print('The scores are:')
    for sc in arg.score:
        print(sc,end=' ')

#如果第一个参数是 None 的类型,就调用这个函数
```

```
def doNothing(arg,verbose=False):
    print('Nothing to do.')
#可以把 register()当作函数使用来注册指定类型
fun.register(type(None),doNothing)

#调用原始函数
fun('Hello world.')
#调用针对整型参数的函数
fun(666,True)
#调用针对实型参数的函数
fun(6.66)
#调用针对列表和元组参数的函数
fun(list(range(5,10)))
fun(tuple(range(10,15)))
#调用针对 None 类型参数的函数 doNothing()
fun(None)
#调用原始函数
fun({1,2,3},True)
#调用针对自定义类型 Scores 参数的函数
fun(Scores(1,2,3,4,5))
```

5.8 协程函数

Python 3.5 之后的版本还引入了一种新的协程函数,使用 async def 进行定义或者使用@ asyncio. coroutine 作为修饰器,如果不需要支持旧版本的 Python,推荐优先使用 async def 定义协程函数。Python 3. 6. x 开始进一步改进了设计,支持在协程函数中同时使用 await 和 yield,这样就可以定义异步生成器对象了。这里先介绍一下协程函数的定义语法和基本用法,更多关于协程的内容请参考 12. 3 节。

下面的代码使用 asyn def 定义了一个协程函数,并调用该函数输出 Hello world。

```
import asyncio

async def hello_world():
    print("Hello World!")

#启动事件循环
loop=asyncio.get_event_loop()
#创建任务,调用函数并等待函数执行结束
loop.run_until_complete(hello_world())
loop.close()
```

下面的代码使用@ asyncio. coroutine 修饰器定义了一个协程函数,并调用该函数连续输出 60 次当前时间。



```
import asyncio
import datetime

@asyncio.coroutine
def display_date(loop):
    end_time=loop.time() +60
    while True:
        print(datetime.datetime.now())
        if (loop.time() +1.0) >=end_time:
            break
        #注意,是 yield from,不是 yield
        yield from asyncio.sleep(1)

loop=asyncio.get_event_loop()
#调用函数并等待函数执行结束
loop.run_until_complete(display_date(loop))
loop.close()
```

其中的协程函数 display_date(loop)也可以使用下面的方式定义：

```
async def display_date(loop):
    end_time=loop.time() +60
    while True:
        print(datetime.datetime.now())
        if (loop.time() +1.0) >=end_time:
            break
        await asyncio.sleep(1)
```

下面的代码定义了两个协程函数,并在一个协程函数中调用另一个协程函数,实现了两者之间的同步。

```
import asyncio

async def compute(x,y):
    print("Computing %s +%s ..." % (x,y))
    await asyncio.sleep(3.0)
    return x +y

async def print_sum(x,y):
    result =await compute(x,y)
    print("Success!\n%s +%s =%s" % (x,y,result))

loop=asyncio.get_event_loop()
loop.run_until_complete(print_sum(1,2))
loop.close()
```

下面的代码使用协程函数定义了一个异步生成器。

```
import asyncio

async def ticker(delay,to):
    for i in range(to):
        yield i
        await asyncio.sleep(delay)

async def run():
    async for i in ticker(1,10):
        print(i)

loop=asyncio.get_event_loop()
try:
    loop.run_until_complete(run())
finally:
    loop.close()
```

5.9 注册程序退出时必须执行的函数

标准库 atexit 支持注册程序退出时执行的函数，下面的程序运行结束时会自动调用 test( ) 函数。

```
import atexit

def test(v):
    print(v)
    print('Exit…')

atexit.register(test,3)
print('test…')
```

程序运行结果为

```
test…
3
Exit…
```

5.10 回调函数原理与实现

回调函数的定义与普通函数并没有本质的区别，区别在于回调函数不是用来直接调用的，而是作为参数传递给另一个函数，当另一个函数中触发了某个事件、满足了某个条件时就会自动调用回调函数。下面的代码演示了回调函数的定义与使用，在删除文件时

如果出现异常则自动调用回调函数,在回调函数中修改文件属性然后再次进行删除。

```
import os
import stat

def remove_readonly(func,path):                #定义回调函数
    os.chmod(path,stat.S_IWRITE)               #删除文件的只读属性
    func(path)                                 #再次调用刚刚失败的函数

def del_dir(path,onerror=None):
    for file in os.listdir(path):
        file_or_dir=os.path.join(path,file)
        if os.path.isdir(file_or_dir) and not os.path.islink(file_or_dir):
            del_dir(file_or_dir)               #递归删除子文件夹及其文件
        else:
            try:
                os.remove(file_or_dir)         #尝试删除该文件
            except:                            #删除失败
                if onerror and callable(onerror):
                    onerror(os.remove,file_or_dir)      #自动调用回调函数
                else:
                    print('You have an exception but did not capture it.')
    os.rmdir(path)                             #删除文件夹

del_dir("E:\\old",remove_readonly)             #调用函数,指定回调函数
```

5.11　精彩案例赏析

示例 5-1　编写函数,接收任意多个实数,返回一个元组,其中第一个元素为所有参数的平均值,其他元素为所有参数中大于平均值的实数。

```
def demo(*para):
    avg=sum(para)/len(para)                    #平均值
    g=[i for i in para if i>avg]               #列表推导式
    return (avg,) +tuple(g)
```

示例 5-2　编写函数,接收字符串参数,返回一个元组,其中第一个元素为大写字母个数,第二个元素为小写字母个数。

```
def demo(s):
    result=[0,0]
    for ch in s:
        if ch.islower():
            result[1] +=1
        elif ch.isupper():
```

```
            result[0] +=1
    return tuple(result)
```

示例 5-3 编写函数,接收包含 n 个整数的列表 lst 和一个整数 k(0≤k＜n)作为参数,返回新列表。处理规则为:将列表 lst 中下标 k 之前的元素逆序,下标 k 之后的元素逆序,然后将整个列表 lst 中的所有元素逆序。

```
def demo(lst,k):
    x=lst[k-1::-1]
    y=lst[:k-1:-1]
    return list(reversed(x+y))
```

本例描述的实际上是将列表循环左移 k 位的算法实现,下面的代码使用了更加直接的方法,但对于长列表来说效率远不如上面的代码高,因为 pop(0)操作在列表首部删除元素,这会引起大量元素的前移。

```
def demo(lst,k):
    temp=lst[:]
    for i in range(k):
        temp.append(temp.pop(0))
    return temp
```

搞清楚问题的本质以后,对于本例中描述的问题,使用切片可以直接实现,可以达到最快的速度。

```
def demo(lst,k):
    return lst[k:] +lst[:k]
```

Python 标准库 collections 提供的双端队列可以直接实现循环左移位和右移位,更加灵活方便。

```
>>>from collections import deque
>>>q=deque(range(20))                    #创建双端队列
>>>q.rotate(3)                           #循环右移位
>>>q
deque([17,18,19,0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16])
>>>q.rotate(-3)                          #循环左移位
>>>q
deque([0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19])
```

示例 5-4 编写函数,接收一个整数 t 作为参数,打印杨辉三角前 t 行。

```
def yanghui(t):
    print([1])
    line=[1,1]
    print(line)
    for i in range(2,t):
        r=[]
```



```
        for j in range(0,len(line)-1):
            r.append(line[j]+line[j+1])
        line=[1]+r+[1]
        print(line)
```

示例 5-5　编写函数，使用 collections 标准库的 defaultdict 实现上例的功能。

杨辉三角也可以使用 Python 标准库 collections 提供的 defaultdict 类来实现，也就是带默认值的字典。如果需要访问 defaultdict 类的对象中某个特定的值但不存在时，不会抛出异常，而是会给出一个默认值。

```
from collections import defaultdict

def yanghui(n):
    #所有元素默认值为 0
    triangle=defaultdict(int)
    for row in range(n):
        #每行第一个元素为 1
        triangle[row,0]=1
        print(triangle[row,0],end='\t')
        #生成该行后续元素
        for col in range(1,row+1):
            #如果指定位置的元素不存在，默认为 0
            triangle[row,col]=triangle[row-1,col-1]+triangle[row-1,col]
            print(triangle[row,col],end='\t')
        print()

yanghui(14)
```

示例 5-6　编写函数，接收一个正偶数作为参数，输出两个素数，并且这两个素数之和等于原来的正偶数。如果存在多组符合条件的素数，则全部输出。

```
def demo(n):
    def IsPrime(p):
        if p==2:
            return True
        if p%2==0:
            return False
        for i in range(3,int(p**0.5)+1,2):
            if p%i==0:
                return False
        return True

    if isinstance(n,int) and n>0 and n%2==0:
        for i in range(2,n//2+1):
            if IsPrime(i) and IsPrime(n-i):
```

```
            print(i,'+',n-i,'=',n)
```

示例 5-7 编写函数,接收两个正整数作为参数,返回一个元组,其中第一个元素为最大公约数,第二个元素为最小公倍数。

```
def demo(m,n):
    p=m*n
    while m%n!=0:
        m,n=n,m%n
    return (n,p//n)
```

另外,Python 标准库 fractions 中提供了 gcd()函数用来计算最大公约数,在 Python 3.5 和更新版本中,标准库 math 也提供了计算最大公约数的函数 gcd()。利用 gcd()函数,上面的代码也可以写作:

```
def demo(m,n):
    import math
    r=math.gcd(m,n)
    return (r,(m*n)//r)
```

示例 5-8 编写函数,接收一个所有元素值都不相等的整数列表 x 和一个整数 n,要求将值为 n 的元素作为支点,将列表中所有值小于 n 的元素全部放到 n 的前面,所有值大于 n 的元素放到 n 的后面。

```
def demo(x,n):
    t1=[i for i in x if i<n]
    t2=[i for i in x if i>n]
    return t1 +[n] +t2
```

上面的代码已经很棒了,只是还有一点小瑕疵。这段代码使用了两个列表推导式,对列表 x 中的元素扫描了两遍。下面的代码虽然看起来长了一点,但是只需要对列表中的元素扫描一遍就能得到结果,对于长列表而言执行效率还是有很大提升的。

```
def demo(x,n):
    t1=[]
    t2=[]
    for i in x:
        if i<n:
            t1.append(i)
        elif i>n:
            t2.append(i)
    return t1 +[n] +t2
```

示例 5-9 编写函数,计算字符串匹配的准确率。

以打字练习程序为例,假设 origin 为原始内容,userInput 为用户输入的内容,下面的代码用来测试用户输入的准确率。



```
def Rate(origin,userInput):
    if not (isinstance(origin,str) and isinstance(userInput,str)):
        print('The two parameters must be strings.')
        return
    right=sum((1 for o,u in zip(origin,userInput) if o==u))
    return round(right/len(origin),2)
```

示例 5-10　编写函数，使用非递归方法对整数进行因数分解。

```
from random import randint
from math import sqrt

def factoring(n):
    '''对大数进行因数分解'''
    if not isinstance(n,int):
        print('You must give me an integer')
        return
    #开始分解，把所有因数都添加到 result 列表中
    result=[]
    for p in primes:
        while n!=1:
            if n%p==0:
                n=n/p
                result.append(p)
            else:
                break
        else:
            result='*'.join(map(str,result))
            return result
    #考虑参数本身就是素数的情况
    if not result:
        return n

testData=[randint(10,100000) for i in range(50)]
#随机数中的最大数
maxData=max(testData)
#小于 maxData 的所有素数
primes=[ p for p in range(2,maxData) if 0 not in
        [ p% d for d in range(2,int(sqrt(p))+1)] ]

for data in testData:
    r=factoring(data)
    print(data,'=',r)
    #测试分解结果是否正确
    print(data==eval(r))
```

示例 5-11　编写函数模拟猜数游戏。系统随机产生一个数，玩家最多可以猜 5 次，系统会根据玩家的猜测进行提示，玩家则可以根据系统的提示对下一次的猜测进行适当调整。

```
from random import randint

def guess(maxValue=100,maxTimes=5):
    #随机生成一个整数
    value=randint(1,maxValue)
    for i in range(maxTimes):
        prompt='Start to Guess:' if i==0 else 'Guess again:'
        #使用异常处理结构，防止输入不是数字的情况
        try:
            x=int(input(prompt))
        except:
            print('Must input an integer between 1 and ',maxValue)
        else:
            #猜对了
            if x==value:
                print('Congratulations!')
                break
            elif x>value:
                print('Too big')
            else:
                print('Too little')
    else:
        #次数用完还没猜对，游戏结束，提示正确答案
        print('Game over. FAIL.')
        print('The value is ',value)
```

示例 5-12　编写函数，计算形式如 a + aa + aaa + aaaa + … + aaa…aaa 的表达式的值，其中 a 为小于 10 的自然数。

```
def demo(v,n):
    assert type(n)==int and 0<v<10,'v must be integer between 1 and 9'
    result,t=0,0
    for i in range(n):
        t=t*10+v
        result+=t
    return result

print(demo(3,4))
```

示例 5-13　编写函数模拟报数游戏。有 n 个人围成一圈，顺序编号，从第一个人开始从 1 到 k（假设 k = 3）报数，报到 k 的人退出圈子，然后圈子缩小，从下一个人继续游戏，问

最后留下的是原来的第几号。

```
from itertools import cycle

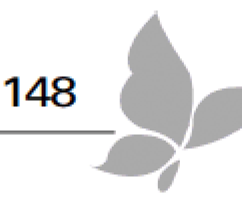
def demo(lst,k):
    #切片,以免影响原来的数据
    t_lst=lst[:]
    #游戏一直进行到只剩下最后一个人
    while len(t_lst) >1:
        #创建 cycle 对象
        c=cycle(t_lst)
        #从 1 到 k 报数
        for i in range(k):
            t=next(c)
        #一个人出局,圈子缩小
        index=t_lst.index(t)
        t_lst=t_lst[index+1:] +t_lst[:index]
    #游戏结束
    return t_lst[0]

lst=list(range(1,11))
print(demo(lst,3))
```

示例 5-14　汉诺塔问题基于递归算法的实现。

据说古代有一个梵塔,塔内有 3 个底座 A、B、C,A 座上有 64 个盘子,盘子大小不等,大的在下,小的在上。有一个和尚想把这 64 个盘子从 A 座移到 C 座,但每次只能允许移动一个盘子,在移动盘子的过程中可以利用 B 座,但任何时刻 3 个座上的盘子都必须始终保持大盘在下、小盘在上的顺序。如果只有一个盘子,则不需要利用 B 座,直接将盘子从 A 移动到 C 即可。和尚想知道这项任务的详细移动步骤和顺序。这实际上是一个非常巨大的工程,是一个不可能完成的任务。根据数学知识我们可以知道,移动 n 个盘子需要 2^n-1 步,64 个盘子需要 18 446 744 073 709 551 615 步。如果每步需要一秒的话,那么就需要 584 942 417 355.072 年。

```
def hannoi(num,src,dst,temp=None):
    #声明用来记录移动次数的变量为全局变量
    global times
    #确认参数类型和范围
    assert type(num)==int,'num must be an integer'
    assert num>0,'num must >0'
    #只剩最后或只有一个盘子需要移动,这也是函数递归调用的结束条件
    if num==1:
        print('The {0} Times move:{1}==>{2}'.format(times,src,dst))
        times +=1
    else:
```



```
        #递归调用函数自身
        #先把除最后一个盘子之外的所有盘子移动到临时柱子上
        hannoi(num-1,src,temp,dst)
        #把最后一个盘子直接移动到目标柱子上
        hannoi(1,src,dst)
        #把除最后一个盘子之外的其他盘子从临时柱子上移动到目标柱子上
        hannoi(num-1,temp,dst,src)
#用来记录移动次数的变量
times=1
#A表示最初放置盘子的柱子,C是目标柱子,B是临时柱子
hannoi(3,'A','C','B')
```

示例 5-15 汉诺塔问题基于非递归算法的实现(感谢国防科技大学刘万伟老师提供本案例算法和第一版本的代码)。

```
def hannoi(n):
    #用来记录移动过程中每个盘子的当前位置
    #初始都在A柱子上,即chr(65+0)
    L=[0]* n
    #n个盘子一共需要移动2^n-1次才能完成
    for i in range(1,2* * n):
        #假设盘子编号分别为0,1,2,…,n-1
        #第i步应该移动的盘子编号
        #正好是i的二进制形式中最后连续的0的个数
        b_i=bin(i)
        j=len(b_i) -b_i.rfind('1') -1
        print('第'+str(i)+'步:移动盘子'+str(j+1),chr(65+L[j]),'->',end=' ')
        #把A、B、C三根柱子摆成三角形
        #把第j个盘子移动到下一根柱子上
        #根据j的奇偶性决定是顺时针移动还是逆时针移动
        L[j]=((L[j]+1)%3 if j%2==0 else (L[j]+2)%3)
        #下一根柱子,这里65是A的ASCII码
        print(chr(65+L[j]))

hannoi(3)
```

示例 5-16 编写函数计算任意位数的黑洞数。黑洞数是指这样的整数:由这个数字每位上的数字组成的最大数减去每位数字组成的最小数仍然得到这个数自身。例如,3位黑洞数是495,因为954 - 459 = 495,4位数字是6174,因为7641 - 1467 = 6174。

```
def main(n):
    '''参数n表示数字的位数,例如n=3时返回495,n=4时返回6174'''
    #待测试数范围的起点和结束值
    start=10**(n-1)
    end=10**n
```

```
    #依次测试每个数
    for i in range(start,end):
        #由这几个数字组成的最大数和最小数
        big=''.join(sorted(str(i),reverse=True))
        little=''.join(reversed(big))
        big,little=map(int,(big,little))
        if big-little==i:
            print(i)
n=4
main(n)
```

示例 5-17　24 点游戏是指随机选取 4 张扑克牌(不包括大小王),然后通过四则运算来构造表达式,如果表达式的值恰好等于 24 就赢一次。下面的代码定义了一个函数用来测试随机给定的 4 个数是否符合 24 点游戏规则,如果符合就输出所有可能的表达式。

```
from random import randint
from itertools import permutations

#4 个数字和 3 个运算符可能组成的表达式形式
exps=('((%s %s %s) %s %s) %s %s',
      '(%s %s %s) %s (%s %s %s)',
      '(%s %s (%s %s %s)) %s %s',
      '%s %s ((%s %s %s) %s %s)',
      '%s %s (%s %s (%s %s %s))')
ops=r'+-*/'

def test24(v):
    result=[]
    #Python 允许函数的嵌套定义
    #这个函数对字符串表达式求值并验证是否等于 24
    def check(exp):
        try:
            #有可能会出现除以 0 异常,所以放到异常处理结构中
            return int(eval(exp))==24
        except:
            return False
    #全排列,枚举 4 个数的所有可能顺序
    for a in permutations(v):
        #查找 4 个数的当前排列能实现 24 的表达式
        t=[exp % (a[0],op1,a[1],op2,a[2],op3,a[3]) for op1 in ops for op2 in ops for op3 in ops for
           exp in exps if check(exp % (a[0],op1,a[1],op2,a[2],op3,a[3]))]
        if t:
            result.append(t)
    return result
```

```
for i in range(20):
    print('=' * 20)
    #生成随机数字进行测试
    lst =[randint(1,14) for j in range(4)]
    r =test24(lst)
    if r:
        print(r)
    else:
        print('No answer for ',lst)
```

示例 5-18 八皇后问题。八皇后问题是高斯提出来的,是一个经典的回溯算法问题,其核心为:在国际象棋棋盘(8 行 8 列)上摆放 8 个皇后,要求 8 个皇后中任意两个都不能位于同一行、同一列或同一斜线上。

```
def isValid(s,col):
    '''这个函数用来检查最后一个皇后的位置是否合法'''
    #当前皇后的行号
    row =len(s)
    #检查当前的皇后们是否有冲突
    for r,c in enumerate(s):
        #如果这一列已有皇后,或者某个皇后与当前皇后的水平与垂直距离相等
        #就表示当前皇后位置不合法,不允许放置
        if c ==col or abs(row -r) ==abs(col -c):
            return False
    return True

def queen(n,s =()):
    '''这个函数返回的结果是每个皇后所在列号'''
    #已是最后一个皇后,保存本次结果
    if len(s) ==n:
        return [s]
    res =[]
    for col in range(n):
        if not isValid(s,col): continue
        for r in queen(n,s +(col,)):
            res.append(r)
    return res

#形式转换,最终结果中包含每个皇后所在的行号和列号
result =[[(r,c) for r,c in enumerate(s)] for s in queen(8)]
#输出合法结果的数量
print(len(result))
#输出所有可能的结果,也就是所有皇后的摆放位置
#结果中每个皇后的位置是一个元组,里面两个数分别是行号和列号
for r in result:
```

```
    print(r)
```

示例 5-19　编写函数,模拟兑换零钱。给定一个数,然后兑换成指定面值的零钱,求解所有可能的兑换方法,每个零钱可以使用多次。

```
def makeChanges(total,changes=(1,2,5,10,20,50,100),result=None):
    if result is None:
        result=[]
    if total==0:
        yield result
    for change in changes:
        #兑换的零钱不能超过总金额,并且每个结果是唯一的
        if change>total or (len(result)>0 and result[-1]<change):
            continue
        for r in makeChanges(total-change,changes,result+[change]):
            yield r
#测试
for way in makeChanges(35):
    print(way)
```

示例 5-20　编写函数,查找序列元素的最大值和最小值。给定一个序列,返回一个元组,其中元组第一个元素为序列最大值,第二个元素为序列最小值。

```
def myMaxMin(iterable):
    '''返回序列的最大值和最小值'''
    tMax=tMin=iterable[0]
    for item in iterable[1:]:
        if item>tMax:
            tMax=item
        elif item<tMin:
            tMin=item

    return (tMax,tMin)
```

示例 5-21　编写函数,模拟内置函数 all()、any()和 zip()。

```
def myAll(iterable):
    '''模拟内置函数 all()'''
    #只要有一个元素等价于 False,返回 False
    for item in iterable:
        if not item:
            return False
    #如果所有元素都等价于 True,返回 True
    return True

def myAny(iterable):
```

```
    '''模拟内置函数 any()'''
    #只要有一个元素等价于 True,返回 True
    for item in iterable:
        if item:
            return True
    #如果所有元素都等价于 False,返回 False
    return False

def myZip(*iterables):
    '''模拟内置函数 zip()'''
    #获取所有迭代对象的最小长度
    min_length = min(map(len, iterables))

    #依次返回所有迭代对象中对应位置上元素组成的元组
    for i in range(min_length):
        yield tuple((it[i] for it in iterables))
```

示例 5-22 编写函数,使用非递归算法实现冒泡排序算法。

```
from random import randint

def bubbleSort(lst, reverse=False):
    length = len(lst)
    for i in range(0, length):
        flag = False
        for j in range(0, length-i-1):
            #比较相邻两个元素大小,并根据需要进行交换
            #默认升序排序
            exp = 'lst[j] > lst[j+1]'
            #如果 reverse=True 则降序排序
            if reverse:
                exp = 'lst[j] < lst[j+1]'
            if eval(exp):
                lst[j], lst[j+1] = lst[j+1], lst[j]
                #flag=True 表示本次扫描发生过元素交换
                flag = True
        #如果一次扫描结束后,没有发生过元素交换,说明已经按序排列
        if not flag:
            break

lst = [randint(1,100) for i in range(20)]
print('Before sort:\n', lst)
bubbleSort(lst, True)
print('After sort:\n', lst)
```

示例 5-23　编写函数，使用递归算法实现冒泡排序算法。

```
from random import randint

def bubbleSort(lst,end=None,reverse=False):
    if end==None:
        length=len(lst)
    else:
        length=end
    if length<=1:
        return
    #flag 用来标记本次扫描过程中是否发生了元素的交换
    flag=False
    for j in range(length-1):
        #比较相邻两个元素的大小，并根据需要进行交换
        #默认升序排序
        exp='lst[j]>lst[j+1]'
        #如果 reverse=True 则降序排序
        if reverse:
            exp='lst[j]<lst[j+1]'
        if eval(exp):
            lst[j],lst[j+1]=lst[j+1],lst[j]
            flag=True
    #如果没有发生元素交换，则表示已按序排列
    if flag==False:
        return
    else:
        #对剩余的元素进行排序
        bubbleSort(lst,length-1,reverse)

#测试
lst=[randint(1,100) for i in range(20)]
print('Before sorted:\n',lst)
#升序排序
bubbleSort(lst)
#降序排序
#bubbleSort(lst,reverse=True)
print('After sorted:\n',lst)
```

示例 5-24　编写函数，模拟选择法排序。

```
def selectSort(lst,reverse=False):
    length=len(lst)
    for i in range(0,length):
        #假设剩余元素中第一个最小或最大
        m=i
        #扫描剩余元素
```



```
        for j in range(i +1,length):
            #如果有更小或更大的,就记录下它的位置
            exp ='lst[j] <lst[m]'
            if reverse:
                exp ='lst[j] >lst[m]'
            if eval(exp):
                m=j
        #如果发现更小或更大的,就交换值
        if m! =i:
            lst[i],lst[m] =lst[m],lst[i]
```

示例 5-25 编写函数,模拟二分法查找。

二分法查找算法非常适合在大量元素中查找指定的元素,要求序列已经排好序(这里假设按从小到大排序),首先测试中间位置上的元素是否为想查找的元素,如果是则结束算法;如果序列中间位置上的元素比要查找的元素小,则在序列的后面一半元素中继续查找;如果中间位置上的元素比要查找的元素大,则在序列的前面一半元素中继续查找。重复上面的过程,不断地缩小搜索范围,直到查找成功或者失败(要查找的元素不在序列中)。

```
def binarySearch(lst,value):
    start =0
    end =len(lst)
    while start <end:
        #计算中间位置
        middle =(start +end) // 2
        #查找成功,返回元素对应的位置
        if value ==lst[middle]:
            return middle
        #在后面一半元素中继续查找
        elif value >lst[middle]:
            start =middle +1
        #在前面一半元素中继续查找
        elif value <lst[middle]:
            end =middle -1
    #查找不成功,返回 False
    return False
from random import randint

lst =[randint(1,50) for i in range(20)]
lst.sort()
print(lst)
result =binarySearch(lst,30)
if result ! =False:
    print('Success,its position is:',result)
else:
```



```
    print('Fail. Not exist.')
```

标准库 bisect 实现了二分法查找和插入的有关功能,其中的 bisect_left( ) 和 bisect_right( ) 方法可以用来定位在一个有序列表中插入指定元素而保持新列表有序的正确位置,如果原列表中已存在要插入的元素,那么 bisect_left( ) 返回已有元素前面紧邻的位置,而 bisect_right( ) 返回已有元素后面紧邻的位置;insort_left( ) 和 insort_right( ) 则直接在正确的位置插入新元素并且保持新列表有序。

```
>>>import bisect
>>>lst =list(range(10))                    #创建列表
>>>lst
[0,1,2,3,4,5,6,7,8,9]
>>>bisect.bisect_left(lst,5)               #获取需要插入的新元素的正确位置
5
>>>bisect.bisect_right(lst,5)
6
>>>bisect.bisect_left(lst,5.5)
6
>>>bisect.bisect_right(lst,5.5)
6
>>>lst.insert(6,5.5)                       #插入新元素
>>>bisect.insort_left(lst,7.9)             #插入新元素
>>>lst
[0,1,2,3,4,5,5.5,6,7,7.9,8,9]
```

示例 5-26　编写函数,使用递归法实现二分法查找。

```
from random import randint

def binarySearch(lst,start,end,value):
    #列表中不存在要查找的元素
    if start >end:
        return False
    #待搜索区间的中间位置和该位置上的值
    mid=(start +end) // 2
    midValue =lst[mid]
    #找到了
    if midValue ==value:
        return mid
    elif midValue >value:
        #在前一半元素中查找
        return binarySearch(lst,start,mid -1,value)
    else:
        #在后一半元素中查找
        return binarySearch(lst,mid +1,end,value)
```

```
lst=[randint(1,50) for i in range(20)]
lst.sort()
print(lst)
result=binarySearch(lst,0,20,30)
if result !=False:
    print('Success,its position is:',result)
else:
    print('Fail. Not exist.')
```

示例 5-27 编写函数,模拟快速排序算法。

```
from random import randint

def quickSort(lst,reverse=False):
    if len(lst) <=1:
        return lst
    #默认使用最后一个元素作为枢点
    pivot=lst.pop()
    first,second=[],[]
    #默认使用升序排序
    exp='x<=pivot'
    #reverse=True 表示降序排列
    if reverse==True:
        exp='x>=pivot'
    for x in lst:
        first.append(x) if eval(exp) else second.append(x)
    #递归调用
    return quickSort(first,reverse) +[pivot] +quickSort(second,reverse)

lst=[randint(1,1000) for i in range(10)]
print(quickSort(lst,True))
```

上面的代码思路非常清晰,不过空间开销比较大,如果使用经典的快速排序算法的话,代码可以像下面这样写:

```
def quickSort(x,start,end):
    if start >=end:
        return

    i=start
    j=end
    #使用第一个元素作为枢点
    key=x[start]

    while i<j:
        #从后向前寻找第一个比指定元素小的元素
        while i<j and x[j]>=key:
```

```
            j -=1
        x[i] =x[j]

        #从前向后寻找第一个比指定元素大的元素
        while i<j and x[i] <=key:
            i +=1
        x[j] =x[i]

    x[i] =key
    quickSort(x,start,i -1)
    quickSort(x,i +1,end)
```

示例 5-28　编写函数,实现侏儒排序算法。

```
def gnomeSort(lst):
    i =0
    length =len(lst)
    while i <length:
        #回头看,如果当前元素比前面的大或者相等,就继续往前走
        if i ==0 or lst[i -1] <=lst[i]:
            i +=1
        else:
            #如果当前元素比前面的小,就交换位置,然后后退一步
            lst[i -1],lst[i] =lst[i],lst[i -1]
            i -=1
```

示例 5-29　编写函数,实现归并排序算法,并进行测试。

```
import random

def mergeSort(seq,reverse =False):
    #把原列表分成两部分
    mid =len(seq) // 2
    left,right =seq[:mid],seq[mid:]

    #根据需要进行递归
    if len(left) >1:
        left =mergeSort(left)
    if len(right) >1:
        right =mergeSort(right)

    #现在前后两部分都已排序
    #进行合并
    temp =[]
    while left and right:
        if left[-1] >=right[-1]:
            temp.append(left.pop())
```

```
        else:
            temp.append(right.pop())
    temp.reverse()
    result=(left or right) +temp

    #根据需要进行逆序
    if reverse:
        i,j=0,len(result) -1
        while i<j:
            result[i],result[j]=result[j],result[i]
            i+=1
            j-=1
    return result

for i in range(100000):
    #生成随机测试数据
    reverse=random.choice((True,False))
    x=[random.randint(1,100) for i in range(20)]
    y=sorted(x,reverse=reverse)
    x=mergeSort(x,reverse)
    if x!=y:
        print('error')
```

示例 5-30 编写函数，使用递归法和回溯法生成不重复数字构成的所有整数。

```
data=tuple(range(10))

def demo1(data,k,s=()):
    '''递归法'''
    if len(s)==k and s[0]!=0:
        print(eval(''.join(map(str,s))))
    else:
        for item in data:
            if item not in s:
                demo1(data,k,s+(item,))

def demo2(data,k,s=()):
    '''回溯法'''
    if len(s)==k and s[0]!=0:
        print(eval(''.join(map(str,s))))
    res=[]
    for item in data:
        if item in s:
            continue
        for r in demo2(data,k,s+(item,)):
            res.append(r)
```

```
    return res

demo1(data,4)
demo2(data,4)
```

借助于 Python 标准库 itertools 中的 permutations( ) 函数可以更直观地解决本例的问题。

```
def demo3(data,k):
    '''使用排列产生任意位数的数字'''
    from itertools import permutations
    r =permutations(data,k)
    for item in r:
        if item[0] !=0:
            print(eval(''.join(map(str,item))))
```

示例 5-31　编写函数,查找给定序列的最长递增子序列。

```
from itertools import combinations
from random import sample

def subAscendingList(lst):
    '''返回最长递增子序列'''
    for length in range(len(lst),0, -1):
        #按长度递减的顺序进行查找和判断
        for sub in combinations(lst,length):
            #判断当前选择的子序列是否为递增顺序
            if list(sub) ==sorted(sub):
                #找到第一个就返回
                return sub

def getList(start =0,end =1000,number =20):
    '''生成随机序列'''
    if number >end -start:
        return None
    return sample(range(start,end),number)

def main():
    lst =getList(number =10)
    if lst:
        print(lst)
        print(subAscendingList(lst))

main()
```

示例 5-32　编写函数,寻找给定序列中相差最小的两个数字。

```
import random
```

```
def getTwoClosestElements(seq):
    #先进行排序,使得相邻元素最接近
    #相差最小的元素必然相邻
    seq=sorted(seq)
    #无穷大
    dif=float('inf')
    #遍历所有元素,两两比较,比较相邻元素的差值
    #使用选择法寻找相差最小的两个元素
    for i,v in enumerate(seq[:-1]):
        d=abs(v - seq[i+1])
        if d<dif:
            first,second,dif=v,seq[i+1],d
    #返回相差最小的两个元素
    return (first,second)

seq=[random.random() for i in range(20)]
print(seq)
print(sorted(seq))
print(getTwoClosestElements(seq))
```

示例 5-33 编写函数,计算给定信息序列的熵。

信息熵可以用来判定指定信源发出的信息的不确定性,信息越是杂乱无章毫无规律,信息熵就越大。如果某信源总是发出完全一样的信息,那么熵为 0,也就是说信息是完全可以提前预测和确定的。

```
from math import log
from random import randint

def informationEntropy(lst):
    #数据总个数
    num=len(lst)
    #每个数据出现的次数
    his=dict()
    for data in lst:
        his[data]=his.get(data,0) +1
    #打印各数据出现的次数,以便核对
    print(his)
    #返回信息熵,其中 x/num 为每个数据出现的频率
    return abs(sum(map(lambda x: x/num * log(x/num,2),his.values())))

#功能测试
for i in range(10):
    lst=[randint(1,5) for i in range(randint(5,30))]
    print('Entropy:',informationEntropy(lst))
    print('=' * 20)
```



```
#没有任何变化的信息序列,这里输出的熵应该为 0
print('Entropy:',informationEntropy([1,1,1,1,1,1]))
```

示例 5-34　编写函数,计算任意字符串信息的哈夫曼编码。

```
from heapq import heapify,heappush,heappop
from itertools import count
from collections import Counter
from random import choice
from string import ascii_letters,digits

def huffman(seq,frq):
    #这里的 count()依次生成 0,1,2,3,4,…
    #主要用来入堆时保持顺序
    num=count()
    #对原始列表进行堆化
    trees=list(zip(frq,num,seq))
    heapify(trees)
    while len(trees)>1:
        #弹出堆中频次最少的两个元素
        #_表示不关心这个值
        fa,_,a=heappop(trees)
        fb,_,b=heappop(trees)
        #合并后生成新节点,重新入堆
        heappush(trees,(fa+fb,next(num),[a,b]))

    #返回建好的二叉树
    return trees[0][-1]

def codes(tree,prefix=''):
    if len(tree)==1:
        #注意,这里不能合并成一个 return (tree,prefix)语句
        yield (tree,prefix)
        return
    #二叉树左边节点编码为 0,右边节点编码为 1
    for bit,child in zip('01',tree):
        #在父节点编码基础上,接上每个节点的编码
        for pair in codes(child,prefix+bit):
            yield pair

def main(seq):
    #统计各字符频次
    global temp
    temp=Counter(seq)
    #这里只是为了让输出结果更直观,但实际上会影响效率
    for item in sorted(temp.items(),key=lambda x: x[1],reverse=True):
```

```
        print(item)
    print('=' * 20)
    #根据各字符出现的频次,生成哈夫曼树
    seq=list(temp.keys())
    frq=[temp[t] for t in seq]
    tree=huffman(seq,frq)
    #根据哈夫曼树,返回各字符编码的生成器对象
    return codes(tree)

letters=ascii_letters +digits
avgLength=0
#生成随机字符串,模拟信源信号
seq=''.join((choice(letters) for i in range(100)))
print(seq+'\n'+'=' * 20)
#按编码长度从小到大输出
#这里只是为了让输出结果更直观,但实际上会影响效率
for item in sorted(main(seq),key=lambda x: len(x[1])):
    print(item)
    avgLength +=temp.get(item[0]) * len(item[1])

#计算并输出平均码长
print(avgLength/len(seq))
```

示例 5-35 编写函数,使用筛选法求解小于指定整数的所有素数。

```
def primes(maxNumber):
    '''筛选法获取小于 maxNumber 的所有素数'''
    #待判断整数
    lst=list(range(3,maxNumber,2))
    #最大整数的平方根
    m=int(maxNumber**0.5)
    for index in range(m):
        current=lst[index]
        #如果当前数字已大于最大整数的平方根,结束判断
        if current>m:
            break
        #对该位置之后的元素进行过滤
        lst[index+1:]=list(filter(lambda x: x%current!=0,lst[index+1:]))
    #2 也是素数
    return [2] +lst

print(primes(1000))
```

示例 5-36 模拟整数乘法的小学竖式计算方法。

```
'''小学整数乘法竖式计算示例
```

```
    12345
  ×) 678
  ---------
    98760
   86415
  74070
  -----------
  8369910
'''

from random import randint

def mul(a,b):
    '''小学竖式两个整数相乘的算法实现'''
    #把两个整数分离开成为各位数字再逆序
    aa=list(map(int,reversed(str(a))))
    bb=list(map(int,reversed(str(b))))

    #n位整数和 m位整数的乘积最多是 n+m位整数
    result=[0]*(len(aa)+len(bb))

    #按小学整数乘法竖式计算两个整数的乘积
    for ia,va in enumerate(aa):
        #c表示进位,初始为 0
        c=0
        for ib,vb in enumerate(bb):
            #Python中内置函数 divmod()可以同时计算整商和余数
            c,result[ia+ib]=divmod(va*vb+c+result[ia+ib],10)
        #最高位的余数应进到更高位
        result[ia+ib+1]=c

    #整理,变成正常结果
    result=int(''.join(map(str,reversed(result))))
    return result

#测试
for i in range(100000):
    a=randint(1,1000)
    b=randint(1,1000)
    r=mul(a,b)
    if r!=a*b:
        print(a,b,r,'error')
```

示例 5-37　使用爬山算法和模拟退火算法分别寻找列表中的最大元素,并比较两种算法的性能优劣。

爬山算法是人工智能算法的一种,特点在于局部择优,所以不一定能够得到全局最优解,尽管效率比较高。使用爬山算法寻找序列最大值的思路是:在能看得到的局部范围内寻找最大值,如果当前元素已经是最大值就结束,如果最大值仍在前面就往前移动到该最大值位置(往上爬),重复上面的过程。如果原始数据的大小和分布类似于双峰图或多峰图的话,那么从一个方向开始爬山的话就可以找到全局最大值,并且能节省一些时间。而另一个方向可能无法找到全局最大值,只能找到局部最大值,除非把“邻域”定义的非常大,但是如果邻域定义的非常大的话会严重影响算法效率。

模拟退火算法可以看作是爬山算法的一种改进,如果前方有更优解就前进,如果没有更优解就以一定概率前进。与简单的爬山算法相比,模拟退火算法有可能跳出局部而得到全局最优解,但也有可能得到更差的解,算法参数的设置非常重要。

```
from random import randint,random

def hillMax(lst,howFar):
    '''爬山算法
    lst:待确定最大值的列表
    howFar:爬山时能看到的"最远方",越大越准确'''
    #由于切片是左闭右开区间,所以 howFar 必须大于 1
    assert howFar >1,'howFar must  >1'

    #从列表第一个元素开始爬
    #如果已经到达最后一个元素,或者已找到局部最大值,结束
    start =0
    ll =len(lst)
    while start  <=ll:
        m=lst[start]
        loc =lst[start +1:start +howFar]
        mm =max(loc)
        if m >mm:
            return m
        else:
            #局部最大数的位置
            mmPos =loc.index(mm)
            start  +=mmPos +1

def simAnnealingMax(lst,howFar):
    '''模拟退火算法,与粗暴的爬山算法相比,有可能跳出局部而获得全局最优解,
    也有可能会因为忽略当前的局部最优解而得到更差的解,参数设置很重要
    lst:待确定最大值的列表
    howFar:爬山时能看到的"最远方",越大越准确'''
    #由于切片是左闭右开区间,所以 howFat 必须大于 1
    assert howFar >1,'parameter "howFar" must  >1'
```

```
    #从列表第一个元素开始爬
    #如果已经到达最后一个元素,或者已找到局部最大值,结束
    start =0
    ll =len(lst)
    #随机走动的次数
    times =1
    while start <=ll:
        #当前局部最优解
        m=lst[start]
        #下一个邻域内的数字
        loc =lst[start +1:start +howFar]
        #如果已处理完所有数据,结束
        if not loc:
            return m
        #下一个邻域的局部最优解及其位置
        mm =max(loc)
        mmPos =loc.index(mm)
        #如果下一个邻域内有更优解,走过去
        if m <=mm:
            start +=mmPos +1
        else:
            #如果下一个邻域内没有更优解,以一定的概率前进或结束
            delta =(m -mm)/(m +mm)
            #随机走动次数越多,对概率要求越低
            if delta <=random()/times:
                start +=mmPos +1
                times +=1
            else:
                return m

#性能测试,比较两种算法优劣,其中的参数 k 的值很重要
for j in range(10):
    win =0
    compareTimes =1000
    for i in range(compareTimes):
        lst =[randint(1,100) for i in range(200)]
        k=3
        if simAnnealingMax(lst,k) >=hillMax(lst,k):
            win +=1
    if win >=compareTimes//2:
        print('win')
```

示例 5-38　计算任意单调曲线在给定区间内的近似长度。

```
def curveLength(xs,func):
```

```
    '''xs:x轴的采样点,越密越准确
     func:曲线方程对应的函数'''
    #函数曲线上的采样点坐标(x,y)
    vs = list(zip(xs,map(func,xs)))
    #返回所有折线段长度(欧几里得距离)之和
    return sum(((v[0] - vs[i+1][0])**2 + (v[1] - vs[i+1][1])**2)**0.5
               for i,v in enumerate(vs[:-1]))

#x轴采样点
xs = list(map(lambda x:x/100,range(200)))
#曲线方程对应的函数,在x的区间上应单调
funcs = {'horizontalLine':lambda x: 3, 'diagonalLine':lambda x: x*2,
         'cubicCurve':lambda x: x**3}
#曲线近似长度
for k,v in funcs.items():
    print(k.ljust(15) + ':',curveLength(xs,v))
```

示例 5-39 编写函数,模拟轮盘抽奖游戏。

轮盘抽奖是比较常见的一种游戏,在轮盘上有一个指针和一些不同颜色、不同面积的扇形,用力转动轮盘,轮盘慢慢停下后依靠指针所处的位置来判定是否中奖以及奖项等级。本例中的函数名和很多变量名使用了中文,这在 Python 3.x 中是完全允许的。

```
from random import random

def 轮盘赌(奖项分布):
    本次转盘读数 = random()
    for k,v in 奖项分布.items():
        if v[0] <= 本次转盘读数 < v[1]:
            return k
#各奖项在轮盘上所占比例
奖项分布 = {'一等奖':(0,0.08),
         '二等奖':(0.08,0.3),
         '三等奖':(0.3,1.0)}

中奖情况 = dict()

for i in range(10000):
    本次战况 = 轮盘赌(奖项分布)
    中奖情况[本次战况] = 中奖情况.get(本次战况,0) + 1

for item in 中奖情况.items():
    print(item)
```

第 6 章



代码复用技术(二):面向对象程序设计

面向对象程序设计(Object Oriented Programming,OOP)的思想主要针对大型软件设计而提出,使得软件设计更加灵活,能够很好地支持代码复用和设计复用,代码具有更好的可读性和可扩展性,大幅度降低了软件开发的难度。面向对象程序设计的一个关键性观念是将数据以及对数据的操作封装在一起,组成一个相互依存、不可分割的整体(对象),不同对象之间通过消息机制来通信或者同步。对于相同类型的对象(instance)进行分类、抽象后,得出共同的特征而形成了类(class),面向对象程序设计的关键就是如何合理地定义这些类并且组织多个类之间的关系。

Python 是面向对象的解释型高级动态编程语言,完全支持面向对象的基本功能,如封装、继承、多态以及对基类方法的覆盖或重写。创建类时用变量形式表示对象特征的成员称为数据成员(attribute),用函数形式表示对象行为的成员称为成员方法(method),数据成员和成员方法统称为类的成员。需要注意的是,Python 中对象的概念很广泛,Python 中的一切内容都可以称为对象,函数也是对象,类也是对象。

6.1 类的定义与使用

6.1.1 基本语法

Python 使用 class 关键字来定义类,class 关键字之后是一个空格,接下来是类的名字,如果派生自其他基类的话则需要把所有基类放到一对括号中并使用逗号分隔,然后是一个冒号,最后换行并定义类的内部实现。类名的首字母一般要大写,当然也可以按照自己的习惯定义类名,但是一般推荐参考惯例来命名,并在整个系统的设计和实现中保持风格一致,这一点对于团队合作非常重要。

```
class Car(object):                         #定义一个类,派生自 object 类
    def infor(self):                       #定义成员方法
        print("This is a car")
```

定义了类之后,就可以用来实例化对象,并通过“对象名.成员”的方式来访问其中的数据成员或成员方法。

```
>>>car=Car()                               #实例化对象
```

```
>>>car.infor()                                    #调用对象的成员方法
This is a car
```

在 Python 中,可以使用内置函数 isinstance( )来测试一个对象是否为某个类的实例,或者使用内置函数 type( )查看对象类型。

```
>>>isinstance(car,Car)
True
>>>isinstance(car,str)
False
>>>type(car)
<class '__main__.Car'>
```

Python 提供了一个关键字 pass,执行的时候什么也不会发生,可以用在类和函数的定义中或者选择结构中,表示空语句。如果暂时没有确定如何实现某个功能,或者提前为以后的软件升级预留一点空间,可以使用关键字 pass 来"占位"。

和定义函数一样,在定义类时,也可以使用三引号为类进行必要的注释。

```
>>>class Test:
    '''This is only a test.'''
    pass
>>>Test.__doc__                                   #查看类的帮助文档
'This is only a test.'
```

6.1.2 type 类

在 Python 中,type 是一个特殊的类,可以看作是所有类型(包括 object)的基类。另外,Python 对象都有一个成员__class__可以查看其所属的类,与内置函数 type( )的返回结果一致;所有 Python 类都有一个成员__bases__,返回包含该类所有基类的元组;Python 类的另一个成员__subclasses__( )可以返回该类型的所有子类。

```
>>>car.__class__                                  #car 是 6.1.1 节中类 Car 的实例
<class '__main__.Car'>
>>>car.__class__.__class__                        #自定义类型的基类是 type
<class 'type'>
>>>x=3
>>>x.__class__
<class 'int'>
>>>x.__class__.__class__                          #整型 int 的基类也是 type
<class 'type'>
>>>x.__class__.__class__.__class__
<class 'type'>
>>>x=object()
>>>x.__class__
<class 'object'>
```

```
>>>x.__class__.__class__                     #object 的基类也是 type
<class 'type'>
>>>def f():
   pass
>>>f.__class__                                #查看对象所属类型
<class 'function'>
>>>type(f)
<class 'function'>
>>>''.__class__
<class 'str'>
>>>().__class__
<class 'tuple'>
>>>[].__class__
<class 'list'>
>>>{}.__class__
<class 'dict'>
>>>{}.__class__.__bases__
(<class 'object'>,)
>>>{}.__class__.__bases__[0].__subclasses__()
                                              #查看所有子类
                                              #这里略去了输出结果
>>>object.__subclasses__()[0]
<class 'str_iterator'>
#下面的代码可以退出 Python 环境,结束程序
>>>[c for c in ().__class__.__bases__[0].__subclasses__()
    if c.__name__=='Quitter'][0](0,0)()
```

6.1.3 定义带修饰器的类

与函数一样,定义类时也可以使用修饰器。假设用户创建文件、修改文件、删除文件、查看文件列表等操作都需要先登录系统才行,每个操作之前都需要确定用户是否已经成功登录,可以使用 Python 扩展库 state 中的修饰器 stateful 来实现。使用 pip 安装扩展库 state,如果下面的代码不能运行的话,需要修改 state 安装文件夹里的__init__. py 文件,把第 16 行的“for i in cls. __dict__. itervalues( ):”改为“for i in cls. __dict__. values( ):”,把第 39 行的“except AttributeError, e:”改为“except AttributeError as e:”,把第 44 行和第 46 行的“raise e”都改为 raise。

```
import state
import os

@state.stateful
class User(object):
    class SignIn(state.State):
```

```
        default=True
        @state.behavior
        def signIn(self,usrName,usrPwd):
            if usrName=='admin' and usrPwd=='admin':
                print('Successfully logged in.')
                #如果成功登录就切换至工作状态
                state.switch(self,User.SignedIn)

    class SignedIn(state.State):
        @state.behavior
        def createFile(self):
            with open('test.txt','w') as fp:
                fp.write('created')

        @state.behavior
        def modifyFile(self):
            if not os.path.exists('test.txt'):
                return
            with open('test.txt','a+') as fp:
                fp.write('ok')

        @state.behavior
        def deleteFile(self):
            if os.path.exists('test.txt'):
                os.remove('test.txt')

        @state.behavior
        def listFile(self):
            for f in os.listdir('.'):
                print(f)

zhang=User()
zhang.signIn('admin','admin')
zhang.listFile()
zhang.createFile()
zhang.modifyFile()
zhang.listFile()
zhang.deleteFile()
```

6.2　数据成员与成员方法

6.2.1　私有成员与公有成员

私有成员在类的外部不能直接访问,一般是在类的内部进行访问和操作,或者在类的外部通过调用对象的公有成员方法来访问,而公有成员是可以公开使用的,既可以在类的内部进行访问,也可以在外部程序中使用。

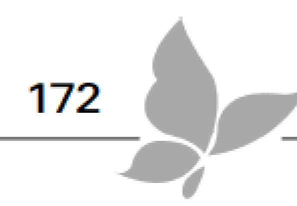
从形式上看,在定义类的成员时,如果成员名以两个(或更多)下画线开头但是不以两个或更多下画线结束则表示是私有成员,否则就不是私有成员。Python 并没有对私有成员提供严格的访问保护机制,通过一种特殊方式“对象名._类名__xxx”也可以在外部程序中访问私有成员,但这会破坏类的封装性,不建议这样做。

```
>>>class A:
    def __init__(self,value1 =0,value2 =0):          #构造方法
        self._value1 =value1
        self.__value2 =value2                        #私有成员
    def setValue(self,value1,value2):                #成员方法,公有成员
        self._value1 =value1
        self.__value2 =value2                        #在类内部可以直接访问私有成员
    def show(self):                                  #成员方法,公有成员
        print(self._value1)
        print(self.__value2)
>>>a =A()
>>>a._value1                                         #在类外部可以直接访问非私有成员
0
>>>a._A__value2                                      #在外部访问对象的私有数据成员
0
```

从严格意义上来讲,这与 Python 的名称绑定机制有关系,在类中以两个或更多下画线开头但不以两个或更多下画线结束的成员绑定到对象时,都会绑定为“对象名._类名__成员名”类似的形式,除非类名中只包含下画线。

```
>>>class Demo:
    def __init__(self,v):
      self.____value =v

>>>d =Demo(3)
>>>d._Demo____value                                  #访问形式被转换
3
>>>class __:
    def __init__(self,v):
      self.____value =v
```




```
>>>dd=__(5)
>>>dd.____value                              #不转换访问形式
5
```

一个圆点“.”是成员访问运算符,可以用来访问命名空间、模块或对象中的成员,在 IDLE、Eclipse + PyDev、WingIDE、PyCharm 或其他 Python 开发环境中,在对象或类名后面加上一个圆点“.”,都会自动列出其所有公开成员,如图 6-1 所示。而如果在圆点“.”后面再加一个下画线,则会列出该对象或类的所有成员,包括私有成员,如图 6-2 所示。当然,也可以使用内置函数 dir( )来查看指定对象、模块或命名空间的所有成员。

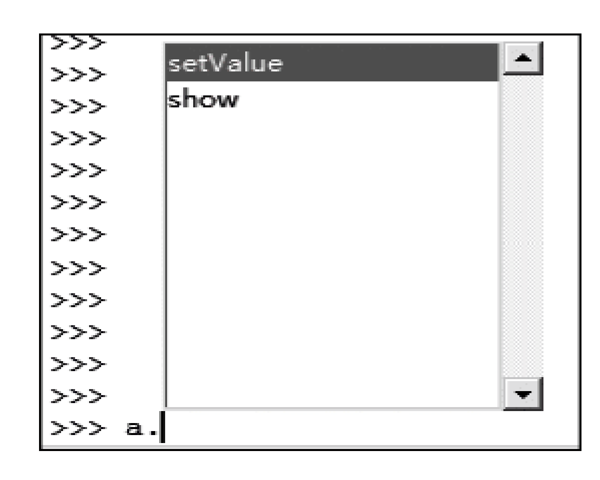


图 6-1　列出对象公开成员

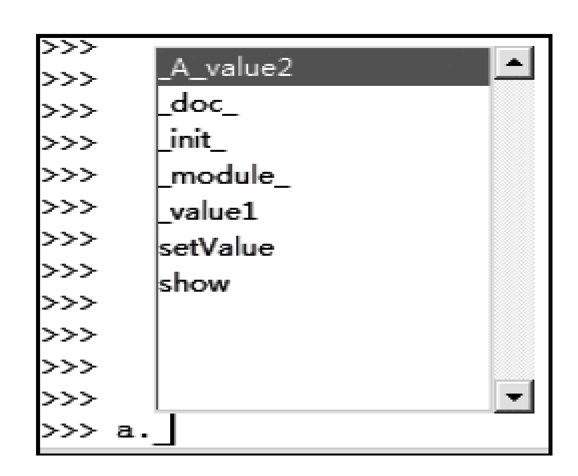


图 6-2　列出对象所有成员

在 Python 中,以下画线开头或结束的成员名有特殊的含义,在类的定义中用下画线作为成员名前缀和后缀往往是表示类的特殊成员。

(1) _xxx:以一个下画线开头,保护成员,只有类对象和子类对象可以访问这些成员,在类的外部一般不建议直接访问;在模块中使用一个或多个下画线开头的成员不能用 from module import * 导入,除非在模块中使用__all__变量明确指明这样的成员可以被导入。

(2) __xxx__:前后各两个下画线,系统定义的特殊成员。

(3) __xxx:以两个或更多下画线开头但不以两个或更多下画线结束,表示私有成员,一般只有类对象自己能访问,子类对象也不能访问该成员,但在对象外部可以通过“对象名._类名__xxx”这样的特殊方式来访问。

6.2.2　数据成员

数据成员可以大致分为两类:属于对象的数据成员和属于类的数据成员。属于对象的数据成员一般在构造方法__init__( )中定义,当然也可以在其他成员方法中定义,在定义和在实例方法中访问数据成员时以 self 作为前缀,同一个类的不同对象(实例)的数据成员之间互不影响;属于类的数据成员是该类所有对象共享的,不属于任何一个对象,在定义类时这类数据成员一般不在任何一个成员方法的定义中。在主程序中或类的外部,对象数据成员属于实例(对象),只能通过对象名访问;而类数据成员属于类,可以通过类名或对象名访问。

利用类数据成员的共享性,可以实时获得该类的对象数量,并且可以控制该类可以创建的对象最大数量。例如:



```
>>>class Demo(object):
    total =0
    def __new__(cls, *args, **kwargs):                    #该方法在__init__()之前被调用
        if cls.total >=3:                                 #最多允许创建 3 个对象
          raise Exception('最多只能创建 3 个对象')
        else:
          return object.__new__(cls)
    def __init__(self):
        Demo.total =Demo.total +1
>>>t1 =Demo()
>>>t1
<__main__.Demo object at 0x00000000034A0278 >
>>>t2 =Demo()
>>>t3 =Demo()
>>>t4 =Demo()
Exception: 最多只能创建 3 个对象
>>>t4
NameError: name 't4' is not defined
```

6.2.3　成员方法、类方法、静态方法、抽象方法

首先应该明确，在面向对象程序设计中，函数和方法这两个概念是有本质区别的。方法一般指与特定实例绑定的函数，通过对象调用方法时，对象本身将被作为第一个参数自动传递过去，普通函数并不具备这个特点。例如，内置函数 sorted( ) 必须要指明要排序的对象，而列表对象的 sort( ) 方法则不需要，默认是对当前列表进行排序。

```
>>>class Demo:
    pass
>>>t =Demo()
>>>def test(self,v):
    self.value =v
>>>t.test =test                                           #动态增加普通函数
>>>t.test
<function test at 0x00000000034B7EA0 >
>>>t.test(t,3)                                            #需要为 self 传递参数
>>>print(t.value)
3
>>>import types
>>>t.test =types.MethodType(test,t)                       #动态增加绑定的方法
>>>t.test
<bound method test of <__main__.Demo object at 0x000000000074F9E8 >>
>>>t.test(5)                                              #不需要为 self 传递参数
>>>print(t.value)
5
```

Python 类的成员方法大致可以分为公有方法、私有方法、静态方法、类方法和抽象方法这几种类型。公有方法、私有方法和抽象方法一般是指属于对象的实例方法,私有方法的名字以两个或更多个下画线开始,而抽象方法一般定义在抽象类中并且要求派生类必须重新实现。每个对象都有自己的公有方法和私有方法,在这两类方法中都可以访问属于类和对象的成员。公有方法通过对象名直接调用,私有方法不能通过对象名直接调用,只能在其他实例方法中通过前缀 self 进行调用或在外部通过特殊的形式来调用。另外,Python 中的类还支持大量的特殊方法,这些方法的两侧各有两个下画线(__),往往与某个运算符或内置函数相对应。

所有实例方法(包括公有方法、私有方法、抽象方法和某些特殊方法)都必须至少有一个名为 self 的参数,并且必须是方法的第一个形参(如果有多个形参的话),self 参数代表当前对象。在实例方法中访问实例成员时需要以 self 为前缀,但在外部通过对象名调用对象方法时并不需要传递这个参数。如果在外部通过类名调用属于对象的公有方法,需要显式为该方法的 self 参数传递一个对象名,用来明确指定访问哪个对象的成员。

静态方法和类方法都可以通过类名和对象名调用,但不能直接访问属于对象的成员,只能访问属于类的成员。另外,静态方法和类方法不属于任何实例,不会绑定到任何实例,当然也不依赖于任何实例的状态,与实例方法相比能够减少很多开销。类方法一般以 cls 作为第一个参数表示该类自身,在调用类方法时不需要为该参数传递值,静态方法则可以不接收任何参数。

```
>>>class Root:
    __total =0
    def __init__(self,v):                  #构造方法,特殊方法
        self.__value =v
        Root.__total +=1

    def show(self):                        #普通实例方法,一般以 self 作为第一个参数的名字
        print('self.__value:',self.__value)
        print('Root.__total:',Root.__total)

    @classmethod                           #修饰器,声明类方法
    def classShowTotal(cls):               #类方法,一般以 cls 作为第一个参数的名字
        print(cls.__total)

    @staticmethod                          #修饰器,声明静态方法
    def staticShowTotal():                 #静态方法,可以没有参数
        print(Root.__total)
>>>r =Root(3)
>>>r.classShowTotal()                      #通过对象来调用类方法
1
>>>r.staticShowTotal()                     #通过对象来调用静态方法
1
>>>rr =Root(5)
```

```
 >>>Root.classShowTotal()                          #通过类名调用类方法
2
 >>>Root.staticShowTotal()                         #通过类名调用静态方法
2
 >>>Root.show()                                    #试图通过类名直接调用实例方法,失败
TypeError: unbound method show() must be called with Root instance as first argument (got nothing
instead)
 >>>Root.show(r)                                   #可以通过这种方法来调用方法并访问实例成员
self.__value: 3
Root.__total: 2
```

抽象方法一般在抽象类中定义,并且要求在派生类中必须重新实现,否则不允许派生类创建实例。

```
import abc

class Foo(metaclass=abc.ABCMeta):                  #抽象类
    def f1(self):                                  #普通实例方法
        print(123)

    def f2(self):                                  #普通实例方法
        print(456)

    @abc.abstractmethod                            #抽象方法
    def f3(self):
        raise Exception('You must reimplement this method.')

class Bar(Foo):
    def f3(self):                                  #必须重新实现基类中的抽象方法
        print(33333)

b=Bar()
b.f3()
```

6.2.4　属　性

公开的数据成员可以在外部随意访问和修改,很难保证用户进行修改时提供新数据的合法性,数据很容易被破坏,也不符合类的封装性要求。解决这一问题的常用方法是定义私有数据成员,然后设计公开的成员方法来提供对私有数据成员的读取和修改操作,修改私有数据成员之前可以对值进行合法性检查,提高了程序的健壮性,保证了数据的完整性。属性是一种特殊形式的成员方法,结合了公开数据成员和成员方法的优点,既可以像成员方法那样对值进行必要的检查,又可以像数据成员一样灵活地访问。

在 Python 3.x 中,属性得到了较为完整的实现,支持更加全面的保护机制。如果设置

属性为只读，则无法修改其值，也无法为对象增加与属性同名的新成员，当然也无法删除对象属性。例如：

```
>>>class Test:
    def __init__(self,value):
        self.__value =value                  #私有数据成员

    @property                               #修饰器，定义属性，提供对私有数据成员的访问
    def value(self):                        #只读属性，无法修改和删除
        return self.__value
>>>t =Test(3)
>>>t.value
3
>>>t.value =5                               #只读属性不允许修改值
AttributeError: can't set attribute
>>>del t.value                              #试图删除对象属性，失败
AttributeError: can't delete attribute
>>>t.value
3
```

下面的代码则把属性设置为可读、可修改，而不允许删除。

```
>>>class Test:
    def __init__(self,value):
        self.__value =value

    def __get(self):                        #读取私有数据成员的值
        return self.__value

    def __set(self,v):                      #修改私有数据成员的值
        self.__value =v

    value =property(__get,__set)            #可读可写属性，指定相应的读写方法

    def show(self):
        print(self.__value)
>>>t =Test(3)
>>>t.value                                  #允许读取属性值
3
>>>t.value =5                               #允许修改属性值
>>>t.value
5
>>>t.show()                                 #属性对应的私有变量也得到了相应的修改
5
>>>del t.value                              #试图删除属性，失败
```

```
AttributeError: can't delete attribute
```

也可以将属性设置为可读、可修改、可删除。

```
>>>class Test:
    def __init__(self,value):
        self.__value=value

    def __get(self):
        return self.__value

    def __set(self,v):
        self.__value=v

    def __del(self):                              #删除对象的私有数据成员
        del self.__value

    value=property(__get,__set,__del)             #可读、可写、可删除的属性

    def show(self):
        print(self.__value)
>>>t=Test(3)
>>>t.show()
3
>>>t.value
3
>>>t.value=5
>>>t.show()
5
>>>t.value
5
>>>del t.value
>>>t.value                                        #相应的私有数据成员已删除,访问失败
AttributeError: 'Test' object has no attribute '_Test__value'
>>>t.show()
AttributeError: 'Test' object has no attribute '_Test__value'
>>>t.value=1                                      #动态增加属性和对应的私有数据成员
>>>t.show()
1
>>>t.value
1
```

6.2.5　类与对象的动态性、混入机制

在 Python 中可以动态地为自定义类和对象增加数据成员和成员方法,这也是 Python

动态类型的一种重要体现。在 6.2.3 节已经见过相关的用法，下面再详细介绍一下。

```
import types
class Car(object):
    price = 100000                                    #属于类的数据成员
    def __init__(self,c):
        self.color = c                                #属于对象的数据成员

car1 = Car("Red")                                     #实例化对象
print(car1.color,Car.price)                           #访问对象和类的数据成员
Car.price = 110000                                    #修改类属性
Car.name = 'QQ'                                       #动态增加类属性
car1.color = "Yellow"                                 #修改实例属性
print(car1.color,Car.price,Car.name)
def setSpeed(self,s):
    self.speed = s
car1.setSpeed = types.MethodType(setSpeed,car1)       #动态为对象增加成员方法
car1.setSpeed(50)                                     #调用对象的成员方法
print(car1.speed)
```

Python 类型的动态性使得我们可以动态为自定义类及其对象增加新的属性和行为，俗称混入（mixin）机制，这在大型项目开发中会非常方便和实用。例如，系统中的所有用户分类非常复杂，不同用户组具有不同的行为和权限，并且可能会经常改变。这时候可以独立地定义一些行为，然后根据需要来为不同的用户设置相应的行为能力。在动画设计中也有类似的技术，例如，可以设计一些动作，然后根据需要把这些动作附加到相应的角色上。

```
>>> import types
>>> class Person(object):
    def __init__(self,name):
        assert isinstance(name,str),'name must be string'
        self.name = name

>>> def sing(self):
    print(self.name + ' can sing.')

>>> def walk(self):
    print(self.name + ' can walk.')

>>> def eat(self):
    print(self.name + ' can eat.')

>>> zhang = Person('zhang')
>>> zhang.sing()                                      #用户不具有该行为
```



```
AttributeError: 'Person' object has no attribute 'sing'
>>>zhang.sing =types.MethodType(sing,zhang)            #动态增加一个新行为
>>>zhang.sing()
zhang can sing.
>>>zhang.walk()
AttributeError: 'Person' object has no attribute 'walk'
>>>zhang.walk =types.MethodType(walk,zhang)
>>>zhang.walk()
zhang can walk.
>>>del zhang.walk                                      #删除用户行为
>>>zhang.walk()
AttributeError: 'Person' object has no attribute 'walk'
```

6.3　继承、多态、依赖注入

6.3.1　继承

设计一个新类时,如果可以继承一个已有的设计良好的类然后进行二次开发,可以大幅度减少开发工作量,并且可以很大程度地保证质量。在继承关系中,已有的、设计好的类称为父类或基类,新设计的类称为子类或派生类。派生类可以继承父类的公有成员,但是不能继承其私有成员。如果需要在派生类中调用基类的方法,可以使用内置函数 super()或者通过“基类名.方法名()”的方式来实现这一目的。

示例 6-1　设计 Person 类,并根据 Person 派生 Teacher 类,分别创建 Person 类与 Teacher 类的对象。

```
#基类必须继承于 object,否则在派生类中将无法使用 super()函数
class Person(object):
    def __init__(self,name ='',age =20,sex ='man'):
        #通过调用方法进行初始化,这样可以对参数进行更好地控制
        self.setName(name)
        self.setAge(age)
        self.setSex(sex)

    def setName(self,name):
        if not isinstance(name,str):
            raise Exception('name must be a string.')
        self.__name =name

    def setAge(self,age):
        if type(age) !=int:
            raise Exception('age must be an integer.')
        self.__age =age
```

```
    def setSex(self,sex):
        if sex not in ('man','woman'):
            raise Exception('sex must be "man" or "woman"')
        self.__sex=sex

    def show(self):
        print(self.__name,self.__age,self.__sex,sep='\n')

#派生类
class Teacher(Person):
    def __init__(self,name='',age=30,sex='man',department='Computer'):
        #调用基类构造方法初始化基类的私有数据成员
        super(Teacher,self).__init__(name,age,sex)
        #也可以这样初始化基类的私有数据成员
        #Person.__init__(self,name,age,sex)
        #初始化派生类的数据成员
        self.setDepartment(department)

    def setDepartment(self,department):
        if type(department) !=str:
            raise Exception('department must be a string.')
        self.__department=department

    def show(self):
        super(Teacher,self).show()
        print(self.__department)

if __name__=='__main__':
    #创建基类对象
    zhangsan=Person('Zhang San',19,'man')
    zhangsan.show()
    print('=' * 30)

    #创建派生类对象
    lisi=Teacher('Li si',32,'man','Math')
    lisi.show()
    #调用继承的方法修改年龄
    lisi.setAge(40)
    lisi.show()
```

Python 支持多继承,如果父类中有相同的方法名,而在子类中使用时没有指定父类名,则 Python 解释器将从左向右按顺序进行搜索,使用第一个匹配的成员。

下面的代码完整地描述了类的继承机制,请认真体会构造方法、私有方法以及普通公开方法的继承原理。

```
>>>class A():
        def __init__(self):
            self.__private()
            self.public()

        def __private(self):
            print('__private() method of A')

        def public(self):
            print('public() method of A')

        def __test(self):
            print('__test() method of A')
>>>class B(A):                          #注意,B类没有构造方法
        def __private(self):
            print('__private() method of B')

        def public(self):
            print('public() method of B')
>>>b=B()                                #创建派生类对象
__private() method of A
public() method of B
>>>b._B__test()                         #派生类没有继承基类中的私有成员方法
AttributeError: 'B' object has no attribute '_B__test'
>>>b._A__test()                         #通过特殊方法可以访问基类中的私有成员方法
__test() method of A
>>>b._A__private()                      #没有严格意义的私有成员
__private() method of A
>>>class C(A):                          #注意,C类有构造方法
    def __init__(self):
        self.__private()
        self.public()

    def __private(self):
        print('__private() method of C')

    def public(self):
        print('public() method of C')
>>>c=C()
__private() method of C
public() method of C
```

6.3.2　多态

多态(polymorphism)是指基类的同一个方法在不同派生类对象中具有不同的表现和

行为。派生类继承了基类的行为和属性之后,还会增加某些特定的行为和属性,同时还可能会对继承来的某些行为进行一定的改变,这都是多态的表现形式,正所谓龙生九子,子子皆不同。通过第 2 章的学习大家已经知道,Python 大多数运算符可以作用于多种不同类型的操作数,并且对于不同类型的操作数往往有不同的表现,这本身就是多态,是通过特殊方法与运算符重载实现的,将在 6.4 节进行详细介绍。下面的代码主要演示通过在派生类中重写基类方法实现多态。

```
>>>class Animal(object):                    #定义基类
    def show(self):
        print('I am an animal.')
>>>class Cat(Animal):                       #派生类,覆盖了基类的 show()方法
    def show(self):
        print('I am a cat.')
>>>class Dog(Animal):                       #派生类
    def show(self):
        print('I am a dog.')
>>>class Tiger(Animal):                     #派生类
    def show(self):
        print('I am a tiger.')
>>>class Test(Animal):                      #派生类,没有覆盖基类的 show()方法
    pass
>>>x=[item() for item in (Animal,Cat,Dog,Tiger,Test)]
>>>for item in x:                           #遍历基类和派生类对象并调用 show()方法
    item.show()
I am an animal.
I am a cat.
I am a dog.
I am a tiger.
I am an animal.
```

6.3.3 依赖注入技术的不同实现方法

依赖注入(Dependency Injection)又称为控制反转(Inversion of Control),主要用来实现不同模块或类之间的解耦,可以根据需要动态地把某种依赖关系注入到对象中,使得模块的设计更加独立。同时,依赖注入也是多态的一种实现方式。常用的依赖注入途径有接口注入、Set 注入和构造注入 3 种。另外,反射也属于比较常用的依赖注入技术,可以根据给定的不同信息创建不同类型的对象。本节就分别介绍一下这几种依赖注入的技术。

1. 接口注入

首先定义一个接口类,然后在继承了该接口的类中实现特定的接口方法,而在接口方法中根据传入参数的不同做出不同的行为。



```
class Itest:                                    #接口
    def injection(self,testClass):
        ''''''

class Test(Itest):                              #继承接口
    def injection(self,testObject):             #实现接口方法
        self.object =testObject

    def show(self):                             #普通方法
        print(self.object)

class A:                                        #类 A 和 B 是测试用的
    pass

class B:
    pass

t =Test()
t.injection(A())                                #传入不同类型的对象
t.show()
t.injection(B())
t.show()
```

上面代码的运行结果为

```
<__main__.A object at 0x00000000033F6B70 >
<__main__.B object at 0x000000000339B470 >
```

2. Set 注入

这种注入方式是通过类本身提供的一个方法用来注入不同类型的对象来设置自身对象和其他对象的依赖关系。

```
class Test:
    def setObject(self,testObject):             #可实现依赖注入
        self.object =testObject

    def show(self):
        print(self.object)

class A:
    pass

class B:
    pass
```



```
t =Test()
t.setObject(A())                                  #传入不同类型的对象
t.show()
t.setObject(B())
t.show()
```

3. 构造注入

这种注入方式是通过在创建类的实例时为构造方法传入不同类型的对象实现的。

```
class Test:
    def __init__(self,testObject):               #通过构造方法实现依赖注入
        self.object =testObject

    def show(self):
        print(self.object)

class A:
    pass

class B:
    pass

t1 =Test(A())                                     #为构造方法传入不同类型的对象
t1.show()
t2 =Test(B())
t2.show()
```

4. 反射

通过反射技术可以根据传入信息(例如类的名字)的不同来创建不同类型的对象,从而实现多态和依赖注入。

```
class Animal:                                     #定义一个类
    def __init__(self,name):
        self.name =name

    def show(self):
        print(self.name)

class Person(Animal):                             #继承 Animal 类,也可以是一个普通的新类
    pass

a =globals()['Animal']('dog')                     #简单形式的反射
```



```
a.show()

p=globals()['Person']('Zhangsan')                #根据类的名字不同来创建不同的对象
p.show()
```

下面的代码演示了反射的另一种方式：

```
class Animal:                                    #定义一个类
    def __init__(self,name):
        self.name=name

    def show(self):
        print(self.name)

class Person(Animal):                            #继承 Animal 类,也可以是一个普通的新类
    pass

def createObject(testClass,name):
    return testClass(name)

a=createObject(Animal,'dog')                     #创建不同类型的对象
a.show()

p=createObject(Person,'Zhangsan')
p.show()
```

6.4　特殊方法与运算符重载

Python 类有大量的特殊方法,其中比较常见的是构造方法和析构方法。Python 中类的构造方法是__init__(),用来为数据成员设置初始值或进行其他必要的初始化工作,在实例化对象时被自动调用和执行。如果用户没有设计构造方法,Python 会提供一个默认的构造方法用来进行必要的初始化工作。Python 中类的析构方法是__del__(),一般用来释放对象占用的资源,在 Python 删除对象和收回对象空间时被自动调用和执行。如果用户没有编写析构方法,Python 将提供一个默认的析构方法进行必要的清理工作。

在 Python 中,除了构造方法和析构方法之外,还有大量的特殊方法支持更多的功能,例如,运算符重载就是通过在类中重写特殊方法实现的。在自定义类时如果重写了某个特殊方法即可支持对应的运算符或内置函数,具体实现什么工作则完全可以由程序员根据实际需要来定义。表 6-1 列出了其中一部分比较常用的特殊成员,完整列表请参考下面的网址：

https://docs.python.org/3/reference/datamodel.html#special-method-names

表 6-1　Python 类的特殊成员



方　法	功 能 说 明
__new__()	类的静态方法，用于确定是否要创建对象
__init__()	构造方法，创建对象时自动调用
__del__()	析构方法，释放对象时自动调用
__add__()	+
__sub__()	-
__mul__()	*
__truediv__()	/
__floordiv__()	//
__mod__()	%
__pow__()	**
__eq__()、__ne__()、__lt__()、__le__()、__gt__()、__ge__()	= =、! =、<、< =、>、> =
__lshift__()、__rshift__()	< <、> >
__and__()、__or__()、__invert__()、__xor__()	&、|、~、^
__iadd__()、__isub__()	+ =、- =，很多其他运算符也有与之对应的复合赋值运算符
__pos__()	一元运算符 +，正号
__neg__()	一元运算符 -，负号
__contains__ ()	与成员测试运算符 in 对应
__radd__()、__rsub__()	反射加法、反射减法，一般与普通加法和减法具有相同的功能，但操作数的位置或顺序相反，很多其他运算符也有与之对应的反射运算符
__abs__()	与内置函数 abs()对应
__bool__()	与内置函数 bool()对应，要求该方法必须返回 True 或 False
__bytes__()	与内置函数 bytes()对应
__complex__()	与内置函数 complex()对应，要求该方法必须返回复数
__dir__()	与内置函数 dir()对应
__divmod__()	与内置函数 divmod()对应
__float__()	与内置函数 float()对应，要求该方法必须返回实数
__hash__()	与内置函数 hash()对应
__int__()	与内置函数 int()对应，要求该方法必须返回整数
__len__()	与内置函数 len()对应

续表

方　法	功 能 说 明
__next__()	与内置函数 next()对应
__reduce__()	提供对 reduce()函数的支持
__reversed__()	与内置函数 reversed()对应
__round__()	对内置函数 round()对应
__str__()	与内置函数 str()对应,要求该方法必须返回 str 类型的数据
__repr__()	打印、转换,要求该方法必须返回 str 类型的数据
__getitem__()	按照索引获取值
__setitem__()	按照索引赋值
__delattr__()	删除对象的指定属性
__getattr__()	获取对象指定属性的值,对应成员访问运算符"."
__getattribute__()	获取对象指定属性的值,如果同时定义了该方法与__getattr__(),那么__getattr__()将不会被调用,除非在__getattribute__()中显式调用__getattr__()或者抛出 AttributeError 异常
__setattr__()	设置对象指定属性的值
__base__	该类的基类
__class__	返回对象所属的类
__dict__	对象所包含的属性与值的字典
__subclasses__()	返回该类的所有子类
__call__()	包含该特殊方法的类的实例可以像函数一样调用
__get__()	定义了这 3 个特殊方法中任何一个的类称为描述符(descriptor),描述符对象一般作为其他类的属性来使用,这 3 个方法分别在获取属性、修改属性值或删除属性时被调用
__set__()	
__delete__()	

6.5　精彩案例赏析

6.5.1　自定义队列

队列是一种特殊的线性表,只允许在队列尾部插入元素和在队列头部删除元素,具有“先入先出”(FIFO)或“后入后出”(LILO)的特点,在多线程编程、作业管理等方面具有重要的应用。

Python 列表对象的 append()方法用于在列表尾部追加元素,pop(0)可以删除并返回列表头部的元素,可以模拟队列结构的操作。

```
>>>x=[]
```

```
>>>x.append(1)                                  #在尾部追加元素,模拟入队操作
>>>x.append(2)
>>>x.append(3)
>>>x
[1,2,3]
>>>x.pop(0)                                     #在头部弹出元素,模拟出队操作
1
>>>x.pop(0)
2
>>>x.pop(0)
3
>>>x
[]
>>>x.pop(0)                                     #空队列弹出头部元素失败,抛出异常
IndexError: pop from empty list
```

从上面的代码可以看出,使用 Python 列表直接模拟队列结构,无法限制队列的大小,并且当列表为空时进行弹出元素的操作会抛出异常。可以对列表进行封装,增加外围代码,自定义队列类来避免这些问题。

示例 6-2 设计自定义双端队列类,模拟入队、出队等基本操作。

```
class myDeque:
    #构造方法,默认队列大小为 10
    def __init__(self,iterable=None,maxlen=10):
        if iterable==None:
            self._content=[]
            self._current=0
        else:
            self._content=list(iterable)
            self._current=len(iterable)
        self._size=maxlen
        if self._size<self._current:
            self._size=self._current

    #析构方法
    def __del__(self):
        del self._content

    #修改队列大小
    def setSize(self,size):
        if size<self._current:
            #如果缩小队列,需要同时删除后面的元素
            for i in range(size,self._current)[::-1]:
                del self._content[i]
```



```
            self._current = size
        self._size = size

    #在右侧入队
    def appendRight(self, v):
        if self._current < self._size:
            self._content.append(v)
            self._current = self._current + 1
        else:
            print('The queue is full')

    #在左侧入队
    def appendLeft(self, v):
        if self._current < self._size:
            self._content.insert(0, v)
            self._current = self._current + 1
        else:
            print('The queue is full')

    #在左侧出队
    def popLeft(self):
        if self._content:
            self._current = self._current - 1
            return self._content.pop(0)
        else:
            print('The queue is empty')

    #在右侧出队
    def popRight(self):
        if self._content:
            self._current = self._current - 1
            return self._content.pop()
        else:
            print('The queue is empty')

    #循环移位
    def rotate(self, k):
        if abs(k) > self._current:
            print('k must <= ' + str(self._current))
            return
        self._content = self._content[-k:] + self._content[:-k]

    #元素翻转
    def reverse(self):
```

```
        self._content=self._content[::-1]

    #显示当前队列中元素的个数
    def __len__(self):
        return self._current

    #使用 print()打印对象时,显示当前队列中的元素
    def __str__(self):
        return 'myDeque(' +str(self._content) +',maxlen=' +str(self._size) +')'

    #直接对象名当作表达式时,显示当前队列中的元素
    __repr__=__str__

    #队列置空
    def clear(self):
        self._content=[]
        self._current=0

    #测试队列是否为空
    def isEmpty(self):
        return not self._content

    #测试队列是否已满
    def isFull(self):
        return self._current==self._size

if __name__=='__main__':
    print('Please use me as a module.')
```

将上面的代码保存为 myDeque. py 文件,并保存在当前文件夹、Python 安装文件夹或 sys. path 列表指定的其他文件夹中,当然也可以使用 append( )方法把该文件所在文件夹添加到 sys. path 列表中。下面的代码演示了自定义队列类的用法:

```
>>>from myDeque import myDeque                      #导入自定义双端队列类
>>>q=myDeque(range(5))                              #创建双端队列对象
>>>q
myDeque([0,1,2,3,4],maxlen=10)
>>>q.appendLeft(-1)                                 #在队列左侧入队
>>>q.appendRight(5)                                 #在队列右侧入队
>>>q
myDeque([-1,0,1,2,3,4,5],maxlen=10)
>>>q.popLeft()                                      #在队列左侧出队
-1
>>>q.popRight()                                     #在队列右侧出队
5
```

```
>>>q.reverse()                                         #元素翻转
>>>q
myDeque([4,3,2,1,0],maxlen=10)
>>>q.isEmpty()                                         #测试队列是否为空
False
>>>q.rotate(-3)                                        #元素循环左移
>>>q
myDeque([1,0,4,3,2],maxlen=10)
>>>q.setSize(20)                                       #改变队列大小
>>>q
myDeque([1,0,4,3,2],maxlen=20)
>>>q.clear()                                           #清空队列元素
>>>q
myDeque([],maxlen=20)
>>>q.isEmpty()
True
```

6.5.2　自定义栈

栈也是一种运算受限的线性表,仅允许在一端进行元素的插入和删除操作,最后入栈的元素最先出栈,最先入栈的元素最后出栈,即“先入后出”(FILO)或“后入先出”(LIFO)。

使用 Python 列表对象提供的 append()、pop()方法也可以模拟栈结构及其基本运算,但是无法限制栈的大小,并且在栈为空时尝试获取其元素时会引发异常。

```
>>>s=[]
>>>s.append(3)                                         #在尾部追加元素,模拟入栈操作
>>>s.append(5)
>>>s.append(7)
>>>s
[3,5,7]
>>>s.pop()                                             #在尾部弹出元素,模拟出栈操作
7
>>>s.pop()
5
>>>s.pop()
3
>>>s.pop()                                             #列表已空,弹出失败
IndexError: pop from empty list
```

如同封装 Python 列表实现自定义队列类一样,也可以对 Python 列表进行封装来模拟栈结构。

示例 6-3　设计自定义栈类,模拟入栈、出栈、判断栈是否为空、是否已满以及改变栈大小等操作。

```
class Stack:
    #构造方法
    def __init__(self,maxlen=10):
        self._content=[]
        self._size=maxlen
        self._current=0

    #析构方法,释放列表控件
    def __del__(self):
        del self._content

    #清空栈中的元素
    def clear(self):
        self._content=[]
        self._current=0

    #测试栈是否为空
    def isEmpty(self):
        return not self._content

    #修改栈的大小
    def setSize(self,size):
        #不允许新的栈大小小于已有元素数量
        if size<self._current:
            print('new size must >=' +str(self._current))
            return
        self._size=size

    #测试栈是否已满
    def isFull(self):
        return self._current==self._size

    #入栈
    def push(self,v):
        if self._current<self._size:
            #在列表尾部追加元素
            self._content.append(v)
            #栈中元素个数加 1
            self._current=self._current +1
        else:
            print('Stack is Full!')

    #出栈
    def pop(self):
```




```
        if self._content:
            #栈中元素个数减 1
            self._current = self._current -1
            #弹出并返回列表尾部元素
            return self._content.pop()
        else:
            print('Stack is empty!')

    def __str__(self):
        return 'Stack(' +str(self._content) +',maxlen=' +str(self._size) +')'

    #复用__str__方法的代码
    __repr__=__str__
```

将代码保存为 myStack.py 文件,下面的代码演示了自定义栈结构的用法。

```
>>>from myStack import Stack                    #导入自定义栈
>>>s =Stack()                                   #创建栈对象
>>>s.push(5)                                    #元素入栈
>>>s.push(8)
>>>s.push('a')
>>>s.pop()                                      #元素出栈
'a'
>>>s.push('b')
>>>s.push('c')
>>>s                                            #查看栈对象
Stack([5,8,'b','c'],maxlen=10)
>>>s.setSize(8)                                 #修改栈大小
>>>s
Stack([5,8,'b','c'],maxlen=8)
>>>s.setSize(3)
new size must >=4
>>>s.clear()                                    #清空栈元素
>>>s.isEmpty()
True
>>>s.setSize(2)
>>>s.push(1)
>>>s.push(2)
>>>s.push(3)
Stack Full!
```

6.5.3　自定义集合

Python 内置了集合类型,支持并集、差集、交集等运算,并且不允许元素重复。本节案例通过封装 Python 列表模拟了集合类,并重写了大量特殊方法,模拟了集合有关的运算。

示例 6-4 自定义集合类。

```
class Set(object):
    def __init__(self,data=None):
        if data==None:
            self.__data=[]
        else:
            if not hasattr(data,'__iter__'):
                #提供的数据不可迭代,实例化失败
                raise Exception('必须提供可迭代的数据类型')
            temp=[]
            for item in data:
                #集合中的元素必须可哈希
                hash(item)
                if not item in temp:
                    temp.append(item)
            self.__data=temp

    #析构方法
    def __del__(self):
        del self.__data

    #添加元素,要求元素必须可哈希
    def add(self,value):
        hash(value)
        if value not in self.__data:
            self.__data.append(value)
        else:
            print('元素已存在,操作被忽略')

    #删除元素
    def remove(self,value):
        if value in self.__data:
            self.__data.remove(value)
            print('删除成功')
        else:
            print('元素不存在,删除操作被忽略')

    #随机弹出并返回一个元素
    def pop(self):
        if not self.__data:
            print('集合已空,弹出操作被忽略')
            return
        import random
```



```
        item=random.choice(self.__data)
        self.__data.remove(item)
        return item

    #运算符重载,集合差集运算
    def __sub__(self,anotherSet):
        if not isinstance(anotherSet,Set):
            raise Exception('类型错误')
        #空集合
        result=Set()
        #如果一个元素属于当前集合而不属于另一个集合,添加
        for item in self.__data:
            if item not in anotherSet.__data:
                result.__data.append(item)
        return result

    #提供方法,集合差集运算,复用上面的代码
    def difference(self,anotherSet):
        return self -anotherSet

    #|运算符重载,集合并集运算
    def __or__(self,anotherSet):
        if not isinstance(anotherSet,Set):
            raise Exception('类型错误')
        result=Set(self.__data)
        for item in anotherSet.__data:
            if item not in result.__data:
                result.__data.append(item)
        return result

    #提供方法,集合并集运算
    def union(self,anotherSet):
        return self | anotherSet

    #&运算符重载,集合交集运算
    def __and__(self,anotherSet):
        if not isinstance(anotherSet,Set):
            raise Exception('类型错误')
        result=Set()
        for item in self.__data:
            if item in anotherSet.__data:
                result.__data.append(item)
        return result
```

```
#运算符重载,集合对称差集
def __xor__(self,anotherSet):
    return (self -anotherSet) | (anotherSet -self)

#提供方法,集合对称差集运算
def symetric_difference(self,anotherSet):
    return self ^ anotherSet

#==运算符重载,判断两个集合是否相等
def __eq__(self,anotherSet):
    if not isinstance(anotherSet,Set):
        raise Exception('类型错误')
    if sorted(self.__data) ==sorted(anotherSet.__data):
        return True
    return False

#>运算符重载,集合包含关系
def __gt__(self,anotherSet):
    if not isinstance(anotherSet,Set):
        raise Exception('类型错误')
    if self !=anotherSet:
        flag1 =True
        for item in self.__data:
            if item not in anotherSet.__data:
                #当前集合中有的元素不属于另一个集合
                flag1 =False
                break
        flag2 =True
        for item in anotherSet.__data:
            if item not in self.__data:
                #另一个集合中有的元素不属于当前集合
                flag2 =False
                break
        if not flag1 and flag2:
            return True
        return False

#>=运算符重载,集合包含关系
def __ge__(self,anotherSet):
    if not isinstance(anotherSet,Set):
        raise Exception('类型错误')
    return self==anotherSet or self >anotherSet

#提供方法,判断当前集合是否为另一个集合的真子集
```

```
    def issubset(self,anotherSet):
        return self <anotherSet

    #提供方法,判断当前集合是否为另一个集合的超集
    def issuperset(self,anotherSet):
        return self >anotherSet

    #提供方法,清空集合中的所有元素
    def clear(self):
        while self.__data:
            del self.__data[-1]
        print('集合已清空')

    #运算符重载,使得集合可迭代
    def __iter__(self):
        return iter(self.__data)

    #运算符重载,支持 in运算符
    def __contains__(self,value):
        return value in self.__data

    #支持内置函数 len()
    def __len__(self):
        return len(self.__data)

    #直接查看该类对象时调用该函数
    def __repr__(self):
        return '{' +str(self.__data)[1:-1] +'}'

    #使用 print()函数输出该类对象时调用该函数
    __str__=__repr__
```

把这个文件保存成 mySet.py,然后就可以像下面的代码这样使用自定义集合类了。

```
>>>from mySet import Set                    #导入自定义集合类
>>>x =Set(range(10))                        #创建集合对象
>>>y =Set(range(8,15))
>>>z =Set([1,2,3,4,5])
>>>x
{0,1,2,3,4,5,6,7,8,9}
>>>y
{8,9,10,11,12,13,14}
>>>z.add(6)                                 #增加元素
>>>z
{1,2,3,4,5,6}
```

```
>>>z.remove(3)                                    #删除指定元素
删除成功
>>>z
{1,2,4,5,6}
>>>y.pop()                                        #随机删除一个元素
11
>>>x -y                                           #差集
{0,1,2,3,4,5,6,7}
>>>x -z
{0,3,7,8,9}
>>>x.difference(y)
{0,1,2,3,4,5,6,7}
>>>x | y                                          #并集
{0,1,2,3,4,5,6,7,8,9,10,12,13,14}
>>>x.union(y)
{0,1,2,3,4,5,6,7,8,9,10,12,13,14}
>>>x & z                                          #交集
{1,2,4,5,6}
>>>x ^ z                                          #对称差集
{0,3,7,8,9}
>>>x.symetric_difference(y)
{0,1,2,3,4,5,6,7,10,12,13,14}
>>>(x -y) | (y -x)
{0,1,2,3,4,5,6,7,10,12,13,14}
>>>x==y                                           #测试两个集合是否相等
False
>>>x>y                                            #测试集合包含关系
False
>>>y>x
False
>>>x>z
True
>>>x >=z
True
>>>z.issubset(x)                                  #测试 z 是否为 x 的子集
True
>>>x.issuperset(z)                                #测试 x 是否为 z 的超集
True
>>>3 in x                                         #测试集合中是否存在某个元素
True
>>>33 in x
False
>>>len(y)                                         #计算集合中元素的个数
6
```

```
>>>y.clear()
集合已清空
>>>y.pop()
集合已空,弹出操作被忽略
```

6.5.4　自定义数组

示例 6-5　自定义一个数字数组类,支持数组与数字之间的四则运算,数组之间的加法运算、内积运算和大小比较,数组元素访问和修改,以及成员测试等功能。

```
class MyArray:
    '''All the elements in this array must be numbers'''
    def __IsNumber(self,n):
        return isinstance(n,(int,float,complex))

    #构造方法,进行必要的初始化
    def __init__(self,*args):
        if not args:
            self.__value=[]
        else:
            for arg in args:
                if not self.__IsNumber(arg):
                    print('All elements must be numbers')
                    return
            self.__value=list(args)

    #析构方法,释放内部封装的列表
    def __del__(self):
        del self.__value

    #重载运算符 +
    #数组中每个元素都与数字 n 相加,或两个数组相加,返回新数组
    #另外几个算术运算可以参考这个方法改写和扩展
    def __add__(self,n):
        if self.__IsNumber(n):
            #数组中所有元素都与数字 n 相加
            b=MyArray()
            b.__value=[item+n for item in self.__value]
            return b
        elif isinstance(n,MyArray):
            #两个等长的数组对应元素相加
            if len(n.__value)==len(self.__value):
                c=MyArray()
                c.__value=[i+j for i,j in zip(self.__value,n.__value)]
```

```
            return c
        else:
            print('Length not equal')
    else:
        print('Not supported')

#重载运算符 -,数组中每个元素都与数字 n 相减,返回新数组
def __sub__(self,n):
    if not self.__IsNumber(n):
        print(' -operating with ',type(n),' and number type is not supported.')
        return
    b=MyArray()
    b.__value=[item-n for item in self.__value]
    return b

#重载运算符 * ,数组中每个元素都与数字 n 相乘,返回新数组
def __mul__(self,n):
    if not self.__IsNumber(n):
        print(' * operating with ',type(n),' and number type is not supported.')
        return
    b=MyArray()
    b.__value=[item* n for item in self.__value]
    return b

#重载运算符/,数组中每个元素都与数字 n 相除,返回新数组
def __truediv__(self,n):
    if not self.__IsNumber(n):
        print(r'/ operating with ',type(n),' and number type is not supported.')
        return
    b=MyArray()
    b.__value=[item/n for item in self.__value]
    return b

#重载运算符//,数组中每个元素都与数字 n 整除,返回新数组
def __floordiv__(self,n):
    if not isinstance(n,int):
        print(n,' is not an integer')
        return
    b=MyArray()
    b.__value=[item//n for item in self.__value]
    return b

#重载运算符%,数组中每个元素都与数字 n 求余数,返回新数组
def __mod__(self,n):
```



```
        if not self.__IsNumber(n):
            print(r'% operating with ',type(n),' and number type is not supported.')
            return
        b=MyArray()
        b.__value=[item% n for item in self.__value]
        return b

    #重载运算符 **,数组中每个元素都与数字 n进行幂计算,返回新数组
    def __pow__(self,n):
        if not self.__IsNumber(n):
            print('** operating with ',type(n),' and number type is not supported.')
            return
        b=MyArray()
        b.__value=[item**n for item in self.__value]
        return b

    def __len__(self):
        return len(self.__value)

    #直接使用该类对象作为表达式来查看对象的值
    def __repr__(self):
        #equivalent to return 'self.__value'
        return repr(self.__value)

    #支持使用 print()函数查看对象的值
    __str__=__repr__

    #追加元素
    def append(self,v):
        if not self.__IsNumber(v):
            print('Only number can be appended.')
            return
        self.__value.append(v)

    #获取指定下标的元素值,支持使用列表或元组指定多个下标
    def __getitem__(self,index):
        length=len(self.__value)
        #如果指定单个整数作为下标,则直接返回元素值
        if isinstance(index,int) and 0<=index<length:
            return self.__value[index]
        #使用列表或元组指定多个整数下标
        elif isinstance(index,(list,tuple)):
            for i in index:
                if not (isinstance(i,int) and 0<=i<length):
```

```
                return 'index error'
            result=[]
            for item in index:
                result.append(self.__value[item])
            return result
        else:
            return 'index error'

    #修改元素值,支持使用列表或元组指定多个下标,同时修改多个元素值
    def __setitem__(self,index,value):
        length=len(self.__value)
        #如果下标合法,则直接修改元素值
        if isinstance(index,int) and 0<=index<length:
            self.__value[index]=value
        #支持使用列表或元组指定多个下标
        elif isinstance(index,(list,tuple)):
            for i in index:
                if not (isinstance(i,int) and 0<=i<length):
                    raise Exception('index error')
            #如果下标和给的值都是列表或元组,并且个数一样,则分别为多个下标的元素修
            改值
            if isinstance(value,(list,tuple)):
                if len(index)==len(value):
                    for i,v in enumerate(index):
                        self.__value[v]=value[i]
                else:
                    raise Exception('values and index must be of the same length')
            #如果指定多个下标和一个普通值,则把多个元素修改为相同的值
            elif isinstance(value,(int,float,complex)):
                for i in index:
                    self.__value[i]=value
            else:
                raise Exception('value error')
        else:
            raise Exception('index error')

    #支持成员测试运算符 in,测试数组中是否包含某个元素
    def __contains__(self,v):
        return v in self.__value

    #模拟向量内积
    def dot(self,v):
        if not isinstance(v,MyArray):
            print(v,' must be an instance of MyArray.')
```




```
                return
            if len(v) !=len(self.__value):
                print('The size must be equal.')
                return
            return sum([i * j for i,j in zip(self.__value,v.__value)])

        #重载运算符 ==,测试两个数组是否相等
        def __eq__(self,v):
            if not isinstance(v,MyArray):
                print(v,' must be an instance of MyArray.')
                return False
            return self.__value==v.__value

        #重载运算符 <,比较两个数组的大小
        def __lt__(self,v):
            if not isinstance(v,MyArray):
                print(v,' must be an instance of MyArray.')
                return False
            return self.__value<v.__value

if __name__=='__main__':
    print('Please use me as a module.')
```

将上面的程序保存为 MyArray.py 文件,可以作为 Python 模块导入并使用其中的数组类。

```
>>>from MyArray import MyArray                    #导入模块中的自定义类
>>>x=MyArray(1,2,3,4,5,6)                         #实例化对象
>>>y=MyArray(6,5,4,3,2,1)
>>>len(x)                                         #返回数组长度,即数组中的元素个数
6
>>>x+5                                            #每个元素加 5,返回新数组
[6,7,8,9,10,11]
>>>x * 3                                          #每个元素乘以 3,返回新数组
[3,6,9,12,15,18]
>>>x.dot(y)                                       #计算两个数组(一维向量)的内积
56
>>>x.append(7)                                    #在数组尾部追加新元素
>>>x
[1,2,3,4,5,6,7]
>>>x.dot(y)
The size must be equal.
>>>x[9]=8                                         #试图修改元素值
Index type error or out of range
>>>x / 2
```

```
[0.5,1.0,1.5,2.0,2.5,3.0,3.5]
>>>x // 2
[0,1,1,2,2,3,3]
>>>x % 3
[1,2,0,1,2,0,1]
>>>x[2]                                   #返回指定位置的元素值
3
>>>'a' in x                               #测试数组中是否包含某个元素
False
>>>3 in x
True
>>>x <y                                   #比较数组的大小
True
>>>x =MyArray(1,2,3,4,5,6)
>>>x +y                                   #两个数组中对应的元素相加,返回新数组
[7,7,7,7,7,7]
>>>x[[2,3,4]]                             #查看多个位置上的元素值
[3,4,5]
>>>x[[2,3]] =[8,9]                        #同时修改多个元素的值
>>>x
[1,2,8,9,5,6]
>>>x[[1,3,5]] =0                          #为多个元素赋值为相同的值
>>>x
[1,0,8,0,5,0]
```

6.5.5 自定义双链表

链表是常用的一种数据结构,每个节点都有左、右两个指针用来指向该节点的前、后节点,支持节点的双向遍历。

示例 6-6 使用 Python 模拟双链表结构。

```
class Node:
    '''节点结构'''
    def __init__(self,data,leftNode =None,rightNode =None):
        #设置当前节点的值和指向下一个节点的指针
        self.data =data
        self.left =leftNode
        self.right =rightNode

    def insertAfter(self,node):
        if node.data =='HEAD':
            print('You cannot make another head node.')
            return
```



```
        #在当前节点后面插入新节点,要考虑当前节点是否为最后一个节点
        node.right=self.right
        if self.right !=None:
            self.right.left=node
        node.left=self
        self.right=node

    def insertBefore(self,node):
        if self.data=='HEAD':
            print('You cannot insert node before the head node.')
            return
        if node.data=='HEAD':
            print('You cannot make another head node.')
            return

        #在当前节点前面插入新节点,要考虑当前节点是否为第一个节点
        node.left=self.left
        if self.left !=None:
            self.left.right=node
        node.right=self
        self.left=node

    def deleteRight(self):
        #删除当前节点之后的节点
        if self.right==None:
            print('Nothing to delete')
            return
        p=self.right
        if p.right !=None:
            p.right.left=self
        self.right=p.right
        #删除节点,释放空间
        del p

    def deleteLeft(self):
        #删除当前节点之前的节点
        p=self.left
        if p.data=='HEAD':
            print('Nothing to delete.')
            return
        self.left=p.left
        p.left.right=self
        #删除节点,释放空间
        del p
```

```
#头节点
head=Node('HEAD')

p=head
for i in range(1,10):
    #依次生成 9 个数字,并创建相应的节点
    #把节点连接到链表的尾部
    n=Node(i)
    p.insertAfter(n)
    p=n

p=head
#遍历链表节点,插入新节点,删除前后节点
while p:
    if p.data==3:
        p.deleteRight()
        p.deleteLeft()
        p.insertBefore(Node(2.5))
        p.insertAfter(Node(3.5))
        break
    else:
        p=p.right

#正向遍历链表并输出每个节点的值
p=head
while p:
    print(p.data,end='-->')
    tail=p
    p=p.right
print('None')

#反向遍历链表并输出每个节点的值
p=tail
while p.data!='HEAD':
    print(p.data,end='-->')
    p=p.left
print('HEAD')
```

6.5.6 自定义常量类

每个类和对象都有一个称为__dict__的字典成员,用来记录该类或对象所拥有的属性。当访问对象属性时,首先会尝试在对象属性中查找,如果找不到就到类属性中查找。Python 内置类型不支持属性的增加,用户自定义类及其对象一般支持属性和方法的增加



与删除。

在下面定义的常量类中,要求对象的成员必须大写,所有成员的值不能相同,并且不允许修改已有成员的值。

```
>>>class Constants:
    def __setattr__(self,name,value):
        assert name not in self.__dict__,'You can not modify ' +name
        assert name.isupper(),'Constant should be uppercase.'
        assert value not in self.__dict__.values(),'Value already exists.'
        self.__dict__[name] =value

>>>t =Constants()
>>>t.R=3                                    #成员不存在,允许添加
>>>t.R=4                                    #成员已存在,不允许修改
AssertionError: You can not modify R
>>>t.G=4
>>>t.g=4                                    #成员必须大写
AssertionError: Constant should be uppercase.
>>>t.B=4                                    #成员的值不允许相同
AssertionError: Value already exists.
```

也可以使用 Python 标准库 collections 提供的 namedtuple 对象来实现类似的功能,不过需要提前定义好属性的名称,不允许添加新属性,也不允许删除已有属性。

```
>>>import collections
>>>Point =collections.namedtuple('Point',['x','y','z'])      #创建一个类
>>>p1 =Point(3,4,5)                         #实例化一个对象
>>>p1.x
3
>>>p1.y
4
>>>p1.z
5
>>>p1.x=30                                  #不允许修改
AttributeError: can't set attribute
>>>p1.t =5                                  #不允许增加新属性
AttributeError: 'Point' object has no attribute 't'
>>>del p1.x                                 #不允许删除属性
AttributeError: can't delete attribute
```

6.5.7　自定义不允许修改值的字典

与 6.5.6 节的思路类似,也可以自定义不允许修改值的字典,这里略去了字典其他方法的代码,可以参考前面几节的代码进行补充和完善。

```
>>>class constDict:
    def __init__(self):
        self.__dict =dict()
    def __getitem__(self,name):
        return self.__dict.get(name,'Not exists')
    def __setitem__(self,name,value):
        assert name not in self.__dict,'Can not modify ' +name
        self.__dict[name] =value

>>>t =constDict()
>>>t['a'] =3
>>>t['a'] =3                                    #不允许修改字典中的值
AssertionError: Can not modify a
>>>t['a']
3
```

6.5.8 自定义支持 with 关键字的类

如果自定义类中实现了特殊方法__enter__( )和__exit__( ),那么该类的对象就可以像内置函数 open( )返回的文件对象一样支持 with 关键字来实现资源的自动管理。

```
class myOpen:
    def __init__(self,fileName,mode='r'):
        self.fp =open(fileName,mode)

    def __enter__(self):
        return self.fp

    def __exit__(self,exceptionType,exceptionVal,trace):
        self.fp.close()

with myOpen('test.txt') as fp:
    print(fp.read())
```

第 7 章



文本处理(一):字符串

在 Python 中,字符串属于不可变有序序列,使用单引号(这是最常用的,或许是因为敲键盘方便)、双引号、三单引号或三双引号作为定界符,并且不同的定界符之间可以互相嵌套。下面几种都是合法的 Python 字符串:

'abc'、'123'、'中国'、"Python"、'''Tom said,"Let's go"'''

除了支持序列通用操作(包括双向索引、比较大小、计算长度、元素访问、切片、成员测试等)以外,字符串类型还支持一些特有的用法,例如字符串格式化、查找、替换、排版等。但由于字符串属于不可变序列,不能直接对字符串对象进行元素增加、修改与删除等操作,切片操作也只能访问其中的元素而无法使用切片来修改字符串中的字符。另外,字符串对象提供的 replace()和 translate()方法以及大量排版方法也不是对原字符串直接进行修改替换,而是返回一个新字符串作为结果。

Python 支持短字符串驻留机制,对于短字符串,将其赋值给多个不同的对象时,内存中只有一个副本,多个对象共享该副本。然而,这一点并不适用于长字符串,也就是说,长字符串不遵守驻留机制,当把一个长字符串赋值给多个变量时,这些变量并不共享相同的内存地址。

```
>>>a='1234'
>>>b='1234'
>>>id(a)==id(b)                          #短字符串支持内存驻留机制
True
>>>a='1234'*50
>>>b='1234'*50
>>>id(a)==id(b)                          #长字符串不支持内存驻留机制
False
>>>id(a[0])==id(b[0])==id(b[4])          #每个字符在内存中仍只有一份
True
```

如果需要判断一个变量是否为字符串,可以使用内置方法 isinstance()或 type()。除了支持 Unicode 编码的 str 类型之外,Python 还支持字节串类型 bytes,str 类型字符串可以通过 encode()方法使用指定的字符串编码格式编码成为 bytes 对象,而 bytes 对象则可以通过 decode()方法使用正确的编码格式解码成为 str 字符串。另外,也可以使用内置函数 str()和 bytes()在这两种类型之间进行转换。

```
>>>type('中国')
<class 'str'>
>>>type('中国'.encode('gbk'))              #编码成字节串,采用 GBK 编码格式
<class 'bytes'>
>>>bytes                                    #bytes 也是 Python 的内置类
<class 'bytes'>
>>>isinstance('中国',str)
True
>>>type('中国')==str
True
>>>type('中国'.encode())==bytes
True
>>>'中国'.encode()                          #默认使用 UFT-8 进行编码
b'\xe4\xb8\xad\xe5\x9b\xbd'
>>>_.decode()                               #默认使用 UFT-8 进行解码
'中国'
>>>bytes('董付国','gbk')
b'\xb6\xad\xb8\xb6\xb9\xfa'
>>>str(_,'gbk')
'董付国'
```

7.1 字符串编码格式简介

最早的字符串编码是美国标准信息交换码 ASCII,仅对 10 个数字、26 个大写英文字母、26 个小写英文字母及一些其他符号进行了编码。ASCII 码采用一个字节来对字符进行编码,最多只能表示 256 个符号。

随着信息技术的发展和信息交换的需要,各国的文字都需要进行编码,不同的应用领域和场合对字符串编码的要求也略有不同,于是又分别设计了多种不同的编码格式,常见的主要有 UTF-8、UTF-16、UTF-32、GB2312、GBK、CP936、base64、CP437 等。UTF-8 对全世界所有国家需要用到的字符进行了编码,以一个字节表示英语字符(兼容 ASCII),以 3 个字节表示中文,还有些语言的符号使用 2 个字节(例如俄语和希腊语符号)或 4 个字节。GB2312 是我国制定的中文编码,使用一个字节表示英语,2 个字节表示中文;GBK 是 GB2312 的扩充,而 CP936 是微软公司在 GBK 基础上开发的编码方式。GB2312、GBK 和 CP936 都是使用 2 个字节表示中文。

不同编码格式之间相差很大,采用不同的编码格式意味着不同的表示和存储形式,把同一字符存入文件时,写入的内容可能会不同,在试图理解其内容时必须了解编码规则并进行正确的解码。如果解码方法不正确就无法还原信息,从这个角度来讲,字符串编码也具有加密的效果。

Python 3.x 完全支持中文字符,默认使用 UTF-8 编码格式,无论是一个数字、英文字母,还是一个汉字,都按一个字符对待和处理。在 Python 3.x 中甚至可以使用中文作为变量名、函数名等标识符,这在示例 5-39 中曾经演示过。

```
>>>import sys
>>>sys.getdefaultencoding()                          #查看默认编码格式
'utf-8'
>>>s='中国山东烟台'
>>>len(s)                                            #字符串长度,或者包含的字符个数
6
>>>s='中国山东烟台 ABCDE'                             #中文与英文字符同样对待,都算一个字符
>>>len(s)
11
>>>姓名='张三'                                        #使用中文作为变量名
>>>print(姓名)                                        #输出变量的值
张三
```

Python 扩展库 chardet 可以用来检测字节串所采用的编码格式,并提供一个可信度以供参考(有时候可能不准确),常用来判断文本文件的编码格式。

```
>>>x='董付国'.encode()
>>>chardet.detect(x)
{'confidence': 0.87625,'encoding': 'utf-8'}
>>>x='董付国'.encode('gbk')
>>>chardet.detect(x)
{'confidence': 0.73,'encoding': 'windows-1252'}
>>>x='董付国'.encode('cp936')
>>>chardet.detect(x)
{'confidence': 0.73,'encoding': 'windows-1252'}
```

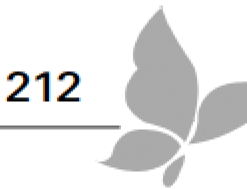
7.2　转义字符与原始字符串

转义字符是指,在字符串中某些特定的符号前加一个斜线之后,该字符将被解释为另外一种含义,不再表示本来的字符。Python 中常用的转义字符如表 7-1 所示。

表 7-1　常用的转义字符

转义字符	含　　义	转义字符	含　　义
\b	退格,把光标移动到前一列位置	\\	一个斜线\
\f	换页符	\'	单引号'
\n	换行符	\"	双引号"
\r	回车	\ooo	3 位八进制数对应的字符
\t	水平制表符	\xhh	2 位十六进制数对应的字符
\v	垂直制表符	\uhhhh	4 位十六进制数表示的 Unicode 字符

下面的代码演示了转义字符的用法:



```
>>>print('Hello\nWorld')                    #包含转义字符的字符串
Hello
World
>>>print('\101')                            #3位八进制数对应的字符
A
>>>print('\x41')                            #2位十六进制数对应的字符
A
>>>print('我是\u8463\u4ed8\u56fd')          #4位十六进制数表示的Unicode字符
我是董付国
```

为了避免对字符串中的转义字符进行转义，可以使用原始字符串，在字符串前面加上字母r或R表示原始字符串，其中的所有字符都表示原始的含义而不会进行任何转义，常用在文件路径、URL和正则表达式等场合。

```
>>>path='C:\Windows\notepad.exe'
>>>print(path)                              #字符\n被转义为换行符
C:\Windows
otepad.exe
>>>path=r'C:\Windows\notepad.exe'           #原始字符串，任何字符都不转义
>>>print(path)
C:\Windows\notepad.exe
```

7.3 字符串格式化

7.3.1 使用%符号进行格式化

使用%符号进行字符串格式化的形式如图7-1所示，格式运算符%之前的部分为格式字符串，之后的部分为需要进行格式化的内容。

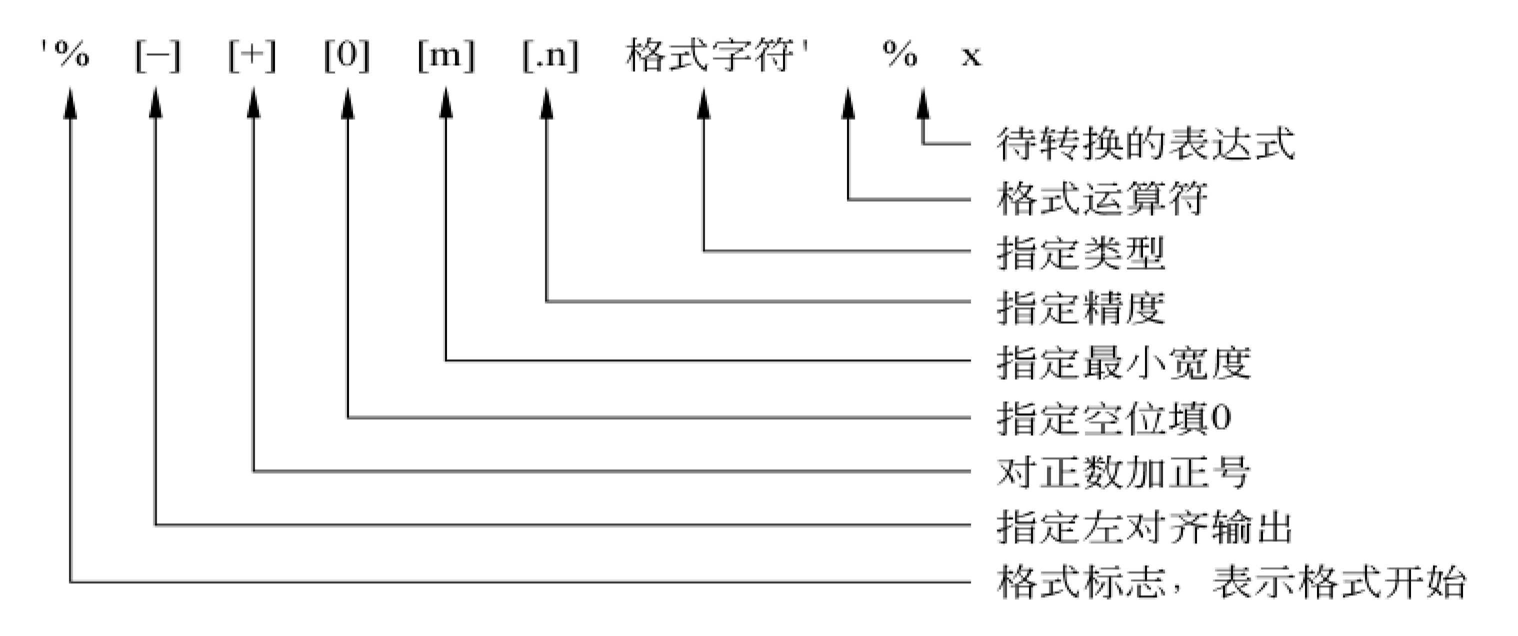


图7-1 字符串格式化

Python支持大量的格式字符，表7-2列出了比较常用的一部分。



表 7-2　格式字符

格式字符	说　　明	格式字符	说　　明
%s	字符串(采用 str()的显示)	%x	十六进制整数
%r	字符串(采用 repr()的显示)	%e	指数(基底写为 e)
%c	单个字符	%E	指数(基底写为 E)
%b	二进制整数	%f、%F	浮点数
%d	十进制整数	%g	指数(e)或浮点数(根据显示长度)
%i	十进制整数	%G	指数(E)或浮点数(根据显示长度)
%o	八进制整数	%%	字符%

使用这种方式进行字符串格式化时,要求被格式化的内容和格式字符之间必须一一对应。

```
>>>x=1235
>>>so="%o" %x
>>>so
'2323'
>>>sh="%x" %x
>>>sh
'4d3'
>>>se="%e" %x
>>>se
'1.235000e+03'
>>>"%s" %65                #等价于 str()
'65'
>>>"%s" %65333
'65333'
>>>'%d,%c' % (65,65)       #使用元组对字符串进行格式化,按位置进行对应
'65,A'
>>>"%d" %"555"             #试图将字符串转换为整数进行输出,抛出异常
TypeError: %d format: a number is required,not str
>>>'%s' % [1,2,3]
'[1,2,3]'
>>>str((1,2,3))            #可以使用 str()函数将任意类型数据转换为字符串
'(1,2,3)'
>>>str([1,2,3])
'[1,2,3]'
```

7.3.2　使用 format()方法进行字符串格式化

除了 7.3.1 节介绍的字符串格式化方法之外,目前 Python 社区更推荐使用 format()

方法进行格式化，该方法非常灵活，不仅可以使用位置进行格式化，还支持使用关键参数进行格式化，更妙的是支持序列解包格式化字符串，为程序员提供了非常大的方便。

在字符串格式化方法 format( )中可以使用的格式主要有 b(二进制格式)、c(把整数转换成 Unicode 字符)、d(十进制格式)、o(八进制格式)、x(小写十六进制格式)、X(大写十六进制格式)、e/E(科学计数法格式)、f/F(固定长度的浮点数格式)、%(使用固定长度浮点数显示百分数)。Python 3.6.x 开始支持在数字常量的中间位置使用单个下画线作为分隔符来提高数字可读性，相应地，字符串格式化方法 format( )也提供了对下画线的支持。下面的代码演示了其中的部分用法：

```
>>>1/3
0.3333333333333333
>>>print('{0:.3f}'.format(1/3))          #保留 3 位小数
0.333
>>>'{0:%}'.format(3.5)                   #格式化为百分数
'350.000000%'
>>>'{0:_},{0:_x}'.format(1000000)        #Python 3.6.0 及更高版本支持
'1_000_000,f_4240'
>>>'{0:_},{0:_x}'.format(10000000)       #Python 3.6.0 及更高版本支持
'10_000_000,98_9680'
>>>print("The number {0:,} in hex is: {0:#x},in oct is {0:#o}".format(55))
The number 55 in hex is: 0x37,in oct is 0o67
>>>print("The number {0:,} in hex is: {0:x},the number {1} in oct is {1:o}".format
(5555,55))
The number 5,555 in hex is: 15b3,the number 55 in oct is 67
>>>print("The number {1} in hex is: {1:#x},the number {0} in oct is {0:#o}".format
(5555,55))
The number 55 in hex is: 0x37,the number 5555 in oct is 0o12663
>>>print("my name is {name},my age is {age},and my QQ is {qq}".format(name = "Dong",qq="306467355",
age=38))
my name is Dong,my age is 38,and my QQ is 306467355
>>>position=(5,8,13)
>>>print("X:{0[0]};Y:{0[1]};Z:{0[2]}".format(position))
                                         #使用元组同时格式化多个值
X:5;Y:8;Z:13
>>>weather=[("Monday","rain"),("Tuesday","sunny"),("Wednesday","sunny"),("Thursday","rain"),("
Friday","cloudy")]
>>>formatter="Weather of '{0[0]}' is '{0[1]}'".format
>>>for item in map(formatter,weather):
    print(item)
```

上面最后一段代码也可以改为下面的写法：

```
>>>for item in weather:
    print(formatter(item))
```

运行结果为

```
Weather of 'Monday' is 'rain'
Weather of 'Tuesday' is 'sunny'
Weather of 'Wednesday' is 'sunny'
Weather of 'Thursday' is 'rain'
Weather of 'Friday' is 'cloudy'
```

7.3.3　格式化的字符串常量

从 Python 3.6.x 开始支持一种新的字符串格式化方式，官方称为 Formatted String Literals，其含义与字符串对象的 format( )方法类似，但形式更加简洁。

```
>>>name='Dong'
>>>age=39
>>>f'My name is {name},and I am {age} years old.'
'My name is Dong,and I am 39 years old.'
>>>width=10
>>>precision=4
>>>value=11/3
>>>f'result:{value:{width}.{precision}}'
'result:      3.667'
```

7.3.4　使用 Template 模板进行格式化

Python 标准库 string 还提供了用于字符串格式化的模板类 Template，可以用于大量信息的格式化，尤其使用于网页模板内容的替换和格式化。例如：

```
>>>from string import Template
>>>t=Template('My name is ${name},and is ${age} years old.')      #创建模板
>>>d={'name':'Dong','age':39}
>>>t.substitute(d)                                              #替换
'My name is Dong,and is 39 years old.'
>>>tt=Template('My name is $name,and is $age years old.')
>>>tt.substitute(d)
'My name is Dong,and is 39 years old.'
>>>html='''<html><head>${head}</head><body>${body}</body></html>'''
>>>t=Template(html)
>>>d={'head':'test','body':'This is only a test.'}
>>>t.substitute(d)
'<html><head>test</head><body>This is only a test.</body></html>'
```

7.4 字符串常用操作

Python 字符串对象提供了大量方法用于字符串的检测、替换和排版等操作，另外还有大量内置函数和运算符也支持对字符串的操作。使用时需要注意的是，字符串对象是不可变的，所以字符串对象提供的涉及字符串“修改”的方法都是返回修改后的新字符串，并不对原字符串做任何修改，无一例外。

7.4.1 find()、rfind()、index()、rindex()、count()

find()和 rfind()方法分别用来查找一个字符串在另一个字符串指定范围(默认是整个字符串)中首次和最后一次出现的位置，如果不存在则返回 -1；index()和 rindex()方法用来返回一个字符串在另一个字符串指定范围中首次和最后一次出现的位置，如果不存在则抛出异常；count()方法用来返回一个字符串在另一个字符串中出现的次数，如果不存在则返回 0。

```
>>>s ="apple,peach,banana,peach,pear"
>>>s.find("peach")              #返回第一次出现的位置
6
>>>s.find("peach",7)            #从指定位置开始查找
19
>>>s.find("peach",7,20)         #在指定范围中进行查找
-1
>>>s.rfind('p')                 #从字符串尾部向前查找
25
>>>s.index('p')                 #返回首次出现的位置
1
>>>s.index('pe')
6
>>>s.index('pear')
25
>>>s.index('ppp')               #指定子字符串不存在时抛出异常
ValueError: substring not found
>>>s.count('p')                 #统计子字符串出现的次数
5
>>>s.count('ppp')               #不存在时返回 0
0
```

一般来说，实际开发时应优先考虑使用 Python 内置函数和内置对象自身提供的方法，运行速度快，并且运行稳定。下面的代码用来检查长字符串中哪些位置上的字母是 a，通过运行结果可以发现，使用字符串对象的 find()方法的速度明显要比逐个字符比较快很多。



```
from string import ascii_letters
from random import choice
from time import time

letters=''.join([choice(ascii_letters) for i in range(999999)])
def positions_of_character(sentence,ch):           #使用字符串对象的 find()方法
    result=[]
    index=0
    index=sentence.find(ch,index+1)
    while index !=-1:
        result.append(index)
        index=sentence.find(ch,index+1)
    return result

def demo(s,c):                                      #普通方法,逐个字符比较
    result=[]
    for i,ch in enumerate(s):                       #也可以使用列表推导式改写
        if ch==c:
            result.append(i)
    return result

for f in (positions_of_character,demo):
    start=time()
    positions=f(letters,'a')
    print(time()-start)
```

运行结果为如下:

```
0.009000539779663086
0.08400487899780273
```

速度居然相差 10 倍左右,这看起来是惊人的。然而,如果把上面代码中

```
letters=''.join([choice(ascii_letters) for i in range(999999)])
```

改为

```
letters=''.join([choice('ab') for i in range(999999)])
```

然后再次运行,会发现结果与上面的代码恰好相反,逐个比较的方法又比使用 find()方法快了很多。代码是完全一样的,只是所查找数据的“密度”不一样,处理速度却有着翻天覆地的变化。综合来说,首先应该分析待处理的数据有什么样的特点(例如数据种类数量、待查找元素的分布密度等),然后才能设计最优的算法和最高效的实现方法。但一般情况下,Python 内置函数、内置对象的方法和标准库对象的效率要高于自己编写的代码。

7.4.2　split()、rsplit()、partition()、rpartition()

字符串对象的 split()和 rsplit()方法分别用来以指定字符为分隔符,从字符串左端和

右端开始将其分隔成多个字符串,并返回包含分隔结果的列表。

```
>>>s="apple,peach,banana,pear"
>>>s.split(",")                          #使用逗号进行分隔
["apple","peach","banana","pear"]
>>>s="2014-10-31"
>>>t=s.split("-")                        #使用指定字符作为分隔符
>>>t
['2014','10','31']
>>>list(map(int,t))                      #将分隔结果转换为整数
[2014,10,31]
```

对于 split( )和 rsplit( )方法,如果不指定分隔符,则字符串中的任何空白符号(包括空格、换行符、制表符等)的连续出现都将被认为是分隔符,返回包含最终分隔结果的列表。

```
>>>s='hello world \n\n My name is Dong '
>>>s.split()
['hello','world','My','name','is','Dong']
>>>s='\n\nhello world \n\n\n My name is Dong '
>>>s.split()
['hello','world','My','name','is','Dong']
>>>s='\n\nhello\t\t world \n\n\n My name\t is Dong '
>>>s.split()
['hello','world','My','name','is','Dong']
```

另外,split( )和 rsplit( )方法允许指定最大分隔次数(注意,并不是必须分隔这么多次)。

```
>>>s='\n\nhello\t\t world \n\n\n My name is Dong '
>>>s.split(maxsplit=1)                   #分隔 1 次
['hello','world \n\n\n My name is Dong ']
>>>s.rsplit(maxsplit=1)
['\n\nhello\t\t world \n\n\n My name is','Dong']
>>>s.split(maxsplit=2)
['hello','world','My name is Dong ']
>>>s.rsplit(maxsplit=2)
['\n\nhello\t\t world \n\n\n My name','is','Dong']
>>>s.split(maxsplit=10)                  #最大分隔次数可以大于实际可分隔次数
['hello','world','My','name','is','Dong']
```

调用 split( )方法如果不传递任何参数,将使用任何空白字符作为分隔符,如果字符串存在连续的空白字符,split( )方法将作为一个空白字符对待。但是,明确传递参数指定 split( )使用的分隔符时,情况略有不同。

```
>>>'a,,,bb,,ccc'.split(',')              #每个逗号都被作为独立的分隔符
```



```
['a','','','bb','','ccc']
>>>'a\t\t\tbb\t\tccc'.split('\t')          #每个制表符都被作为独立的分隔符
['a','','','bb','','ccc']
>>>'a\t\t\tbb\t\tccc'.split()              #连续多个制表符被作为一个分隔符
['a','bb','ccc']
```

字符串对象的 partition( ) 和 rpartition( ) 方法以指定字符串为分隔符将原字符串分隔为 3 部分,即分隔符之前的字符串、分隔符字符串和分隔符之后的字符串。如果指定的分隔符不在原字符串中,则返回原字符串和两个空字符串。如果字符串中有多个分隔符,那么 partition( ) 把从左往右遇到的第一个分隔符作为分隔符,rpartition( ) 把从右往左遇到的第一个分隔符作为分隔符。

```
>>>s="apple,peach,banana,pear"
>>>s.partition(',')                        #从左侧使用逗号进行切分
('apple',',','peach,banana,pear')
>>>s.rpartition(',')                       #从右侧使用逗号进行切分
('apple,peach,banana',',','pear')
>>>s.rpartition('banana')                  #使用字符串作为分隔符
('apple,peach,','banana',',pear')
>>>'abababab'.partition('a')
('','a','bababab')
>>>'abababab'.rpartition('a')
('ababab','a','b')
```

7.4.3 join()

字符串的 join( ) 方法用来将列表中多个字符串进行连接,并在相邻两个字符串之间插入指定字符,返回新字符串。

```
>>>li=["apple","peach","banana","pear"]
>>>sep=","
>>>sep.join(li)                    #使用逗号作为连接符
"apple,peach,banana,pear"
>>>':'.join(li)                    #使用冒号作为连接符
'apple:peach:banana:pear'
>>>''.join(li)                     #使用空字符作为连接符
'applepeachbananapear'
```

使用 split( ) 和 join( ) 方法可以删除字符串中多余的空白字符,如果有连续多个空白字符,只保留一个。

```
>>>x='aaa        bb        c d e   fff      '
>>>' '.join(x.split())             #使用空格作为连接符
'aaa bb c d e fff'
>>>def equavilent(s1,s2):          #判断两个字符串在 Python 意义上是否等价
```

```
    if s1==s2:
        return True
    elif ' '.join(s1.split())==' '.join(s2.split()):
        return True
    elif ''.join(s1.split())==''.join(s2.split()):
        return True
    else:
        return False
>>>equavilent('pip list','pip    list')
True
>>>equavilent('[1,2,3]','[1,2,3]')        #判断两个列表写法是否等价
True
>>>equavilent('[1,2,3]','[1,2 ,3]')
True
>>>equavilent('[1,2,3]','[1,2 ,3 ,4]')
False
```

7.4.4 lower()、upper()、capitalize()、title()、swapcase()

这几个方法分别用来将字符串转换为小写、大写字符串，将字符串首字母变为大写，将每个单词的首字母变为大写以及大小写互换，生成新字符串，不对原字符串做任何修改。

```
>>>s="What is Your Name?"
>>>s.lower()                    #返回小写字符串
'what is your name?'
>>>s.upper()                    #返回大写字符串
'WHAT IS YOUR NAME?'
>>>s.capitalize()               #字符串首字母大写
'What is your name?'
>>>s.title()                    #每个单词的首字母大写
'What Is Your Name?'
>>>s.swapcase()                 #大小写互换
'wHAT IS yOUR nAME?'
```

7.4.5 replace()、maketrans()、translate()

字符串方法 replace()用来替换字符串中指定字符或子字符串的所有重复出现，每次只能替换一个字符或一个字符串，把指定的字符串参数作为一个整体对待，类似于 Word、WPS、记事本等文本编辑器的查找与替换功能。该方法并不修改原字符串，而是返回一个新字符串。

```
>>>s="中国,中国"
>>>print(s.replace("中国","中华人民共和国"))
```



```
中华人民共和国,中华人民共和国
>>>print('abcdabc'.replace('abc','ABC'))
ABCdABC
```

字符串对象的 maketrans( )方法用来生成字符映射表,而 translate( )方法用来根据映射表中定义的对应关系转换字符串并替换其中的字符,使用这两个方法的组合可以同时处理多个不同的字符,replace( )方法则无法满足这一要求。

```
#创建映射表,将字符"abcdef123"一一对应地转换为"uvwxyz@#$"
>>>table=''.maketrans('abcdef123','uvwxyz@#$')
>>>s="Python is a greate programming language. I like it!"
#按映射表进行替换
>>>s.translate(table)
'Python is u gryuty progrumming lunguugy. I liky it!'
```

下面的代码使用 maketrans( )和 translate( )方法实现了恺撒加密算法,其中 k 表示算法密钥,也就是把每个英文字母变为其后面的第几个字母。

```
>>>import string
>>>def kaisa(s,k):
    lower=string.ascii_lowercase          #小写字母
    upper=string.ascii_uppercase          #大写字母
    before=string.ascii_letters
    after=lower[k:] +lower[:k] +upper[k:] +upper[:k]
    table=''.maketrans(before,after)      #创建映射表
    return s.translate(table)

>>>s="Python is a greate programming language. I like it!"
>>>kaisa(s,3)
'Sbwkrq lv d juhdwh surjudpplqj odqjxdjh. L olnh lw!'
>>>s='If the implementation is easy to explain,it may be a good idea.'
>>>kaisa(s,3)
'Li wkh lpsohphqwdwlrq lv hdvb wr hasodlq,lw pdb eh d jrrg lghd.'
```

7.4.6　strip()、rstrip()、lstrip()

这几个方法分别用来删除两端、右端或左端连续的空白字符或指定字符。

```
>>>s=" abc  "
>>>s2=s.strip()                        #删除空白字符
>>>s2
"abc"
>>>'\n\nhello world \n\n'.strip()      #删除空白字符
'hello world'
>>>"aaaassddf".strip("a")              #删除指定字符
"ssddf"
```

```
>>>"aaaassddf".strip("af")
"ssdd"
>>>"aaaassddfaaa".rstrip("a")           #删除字符串右端指定字符
'aaaassddf'
>>>"aaaassddfaaa".lstrip("a")           #删除字符串左端指定字符
'ssddfaaa'
```

这 3 个函数的参数指定的字符串并不作为一个整体对待，而是在原字符串的两侧、右侧、左侧删除参数字符串中包含的所有字符，一层一层地从外往里扒。

```
>>>'aabbccddeeeffg'.strip('af')         #字母 f 不在字符串两侧,所以不删除
'bbccddeeeffg'
>>>'aabbccddeeeffg'.strip('gaf')
'bbccddeee'
>>>'aabbccddeeeffg'.strip('gaef')
'bbccdd'
>>>'aabbccddeeeffg'.strip('gbaef')
'ccdd'
>>>'aabbccddeeeffg'.strip('gbaefcd')
''
```

7.4.7 startswith()、endswith()

这两个方法用来判断字符串是否以指定字符串开始或结束，可以接收两个整数参数来限定字符串的检测范围。

```
>>>s ='Beautiful is better than ugly.'
>>>s.startswith('Be')                   #检测整个字符串
True
>>>s.startswith('Be',5)                 #指定检测范围的起始位置
False
>>>s.startswith('Be',0,5)               #指定检测范围的起始和结束位置
True
```

另外，这两个方法还可以接收一个字符串元组作为参数来表示前缀或后缀，例如，下面的代码可以列出指定文件夹下所有扩展名为 bmp、jpg 或 gif 的图片。

```
>>>import os
>>>[filename for filename in os.listdir(r'D:\\') if filename.endswith(('.bmp','.jpg','.gif'))]
```

7.4.8 isalnum()、isalpha()、isdigit()、isdecimal()、isnumeric()、isspace()、isupper()、islower()

用来测试字符串是否为数字或字母、是否为字母、是否为数字字符、是否为空白字符、是否为大写字母以及是否为小写字母。

```
>>>'1234abcd'.isalnum()
True
>>>'1234abcd'.isalpha()             #全部为英文字母时返回 True
False
>>>'1234abcd'.isdigit()             #全部为数字时返回 True
False
>>>'abcd'.isalpha()
True
>>>'1234.0'.isdigit()
False
>>>'1234'.isdigit()
True
>>>'九'.isnumeric()                 #isnumeric()方法支持汉字数字
True
>>>'九'.isdigit()
False
>>>'九'.isdecimal()
False
>>>'ⅣⅢⅩ'.isdecimal()
False
>>>'ⅣⅢⅩ'.isdigit()
False
>>>'ⅣⅢⅩ'.isnumeric()               #支持罗马数字
True
```

另外,Python 标准库 unicodedata 还提供了不同形式数字字符到十进制数字的转换方法。

```
>>>import unicodedata
>>>unicodedata.numeric('2')
2.0
>>>unicodedata.numeric('九')        #汉字数字
9.0
>>>unicodedata.numeric('Ⅹ')         #罗马数字
10.0
```

7.4.9　center()、ljust()、rjust()、zfill()

这几个方法用于对字符串进行排版,其中 center()、ljust()、rjust()返回指定宽度的新字符串,原字符串居中、左对齐或右对齐出现在新字符串中,如果指定的宽度大于字符串长度,则使用指定的字符(默认是空格)进行填充。zfill()返回指定宽度的字符串,在左侧以字符 0 进行填充。

```
>>>'Hello world!'.center(20)        #居中对齐,以空格进行填充
```

```
'    Hello world!    '
>>>'Hello world!'.center(20,'=')          #居中对齐,以字符=进行填充
'====Hello world!===='
>>>'Hello world!'.ljust(20,'=')           #左对齐
'Hello world!========'
>>>'Hello world!'.rjust(20,'=')           #右对齐
'========Hello world!'
>>>'abc'.zfill(5)                         #在左侧填充数字字符 0
'00abc'
>>>'abc'.zfill(2)                         #指定宽度小于字符串长度时,返回字符串本身
'abc'
>>>'uio'.zfill(20)
'00000000000000000uio'
```

7.4.10 字符串对象支持的运算符

Python 支持使用运算符“+”连接字符串,但该运算符涉及大量数据的复制,效率非常低,不适合大量长字符串的连接。下面的代码演示了运算符“+”和字符串对象 join()方法之间的速度差异。

```
import timeit

#使用列表推导式生成 10000 个字符串
strlist =['This is a long string that will not keep in memory.' for n in range(10000)]

#使用字符串对象的 join()方法连接多个字符串
def use_join():
    return ''.join(strlist)

#使用运算符“+”连接多个字符串
def use_plus():
    result =''
    for strtemp in strlist:
        result =result +strtemp
    return result

if __name__=='__main__':
    #重复运行次数
    times =1000
    jointimer =timeit.Timer('use_join()','from __main__ import use_join')
    print('time for join:',jointimer.timeit(number =times))
    plustimer =timeit.Timer('use_plus()','from __main__ import use_plus')
    print('time for plus:',plustimer.timeit(number =times))
```



该代码分别使用 join( ) 函数和“ + ”对 10000 个字符串进行连接，并重复运行 1000 次，然后输出每种方法所使用的时间，运行结果为

```
time for join: 0.11133914429192587
time for plus: 1.6754796186748913
```

修改上面代码中的参数会发现，随着字符串数量的增多，运算符“ + ”效率会越来越低，并且会产生大量的垃圾数据，严重的话会造成大量内存碎片而影响系统的运行。

另外，timeit 模块还支持下面代码演示的用法，从运行结果可以看出，当需要对大量数据进行类型转换时，内置函数 map( ) 可以提供非常高的效率。

```
>>>import timeit
>>>timeit.timeit('"-".join(str(n) for n in range(100))',number=10000)
                                              #重复运行 10000 次
0.3063435900577929
>>>timeit.timeit('"-".join([str(n) for n in range(100)])',number=10000)
0.27191914957273866
>>>timeit.timeit('"-".join(map(str,range(100)))',number=10000)
0.21119518171659024
```

第 2 章曾经介绍过，Python 字符串支持与整数的乘法运算，表示序列重复，也就是字符串内容的重复。

```
>>>'abcd' * 3
'abcdabcdabcd'
```

最后，与列表、元组、字典、集合一样，也可以使用成员测试运算符 in 来判断一个字符串是否出现在另一个字符串中，返回 True 或 False。

```
>>>"a" in "abcde"                 #测试一个字符中是否存在于另一个字符串中
True
>>>'ac' in 'abcde'                #关键字 in 左边的字符串作为一个整体对待
False
>>>"j" not in "abcde"
True
```

几乎所有论坛或社区都会对用户提交的输入进行检查，并过滤一些非法的敏感词，这极大地促进了网络文明和净化。这样的功能可以使用关键字 in 和 replace( ) 方法来实现。下面的代码用来检测用户输入中是否有不允许的敏感字词，如果有就提示非法。

```
>>>words =('测试','非法','暴力')
>>>text =input('请输入：')
请输入：这句话里含有非法内容
>>>for word in words:
    if word in text:
        print('非法')
        break
```

```
else:
  print('正常')
```

下面的代码则可以用来测试用户输入中是否有敏感词,如果有的话就把敏感词替换为 3 个星号＊＊＊。

```
>>>words=('测试','非法','暴力','话')
>>>text='这句话里含有非法内容'
>>>for word in words:
    if word in text:
        text=text.replace(word,'＊＊＊')
>>>text
'这句＊＊＊里含有＊＊＊内容'
```

或者直接使用正则表达式模块 re 提供的函数进行检查和过滤:

```
>>>words=('测试','非法','暴力','话')
>>>text='这句话里含有非法内容'
>>>import re
>>>re.sub('|'.join(words),'＊＊＊',text)
'这句＊＊＊里含有＊＊＊内容'
```

7.4.11 适用于字符串对象的内置函数

除了字符串对象提供的方法以外,很多 Python 内置函数也可以对字符串进行操作。

```
>>>x='Hello world.'
>>>len(x)                          #字符串长度
12
>>>max(x)                          #最大字符
'w'
>>>min(x)
' '
>>>list(zip(x,x))                  #zip()也可以作用于字符串
[('H','H'),('e','e'),('l','l'),('l','l'),('o','o'),(' ',' '),('w','w'),('o','o'),('r','r'),('l','l'),('d','d'),('.','.')]
>>>sorted(x)
[' ','.','H','d','e','l','l','l','o','o','r','w']
>>>list(reversed(x))
['.','d','l','r','o','w',' ','o','l','l','e','H']
>>>list(enumerate(x))
[(0,'H'),(1,'e'),(2,'l'),(3,'l'),(4,'o'),(5,' '),(6,'w'),(7,'o'),(8,'r'),(9,'l'),(10,'d'),(11,'.')]
>>>list(map(lambda i,j: i+j,x,x))
['HH','ee','ll','ll','oo','  ','ww','oo','rr','ll','dd','..']
```

内置函数 eval()用来把任意字符串转化为 Python 表达式并进行求值。

```
>>>eval("3+4")                    #计算表达式的值
7
>>>a=3
>>>b=5
>>>eval('a+b')                    #这时候要求变量 a 和 b 已存在
8
>>>import math
>>>eval('math.sqrt(3)')
1.7320508075688772
```

在 Python 3.x 中，input()将用户的输入一律按字符串对待，如果需要将其还原为本来的类型，对于简单的整数、实数、复数可以直接使用 int()、float()和 complex()函数进行转换，而对于列表、元组或其他复杂结构则需要使用内置函数 eval()，不能使用 list()、tuple()直接进行转换。不管是使用 int()、float()和 complex()函数还是 eval()函数，在转换时最好配合异常处理结构，以避免因为数据类型不符合要求或者无法正确求值而抛出异常。

```
>>>x=input()
357
>>>x
'357'
>>>eval(x)
357
>>>x=input()
[3,5,7]
>>>x
'[3,5,7]'
>>>eval(x)                        #注意，这里不能使用 list(x)进行转换
[3,5,7]
>>>x=input()
abc
>>>x
'abc'
>>>try:                           #当前作用域中不存在变量 abc
    print(eval(x))
except:
    print('wrong input')

wrong input
```

Python 的内置函数 eval()可以计算任意合法表达式的值，如果有恶意用户巧妙地构造并输入非法字符串，可以执行任意外部程序或者实现其他目的，例如，下面的代码运行后可以启动记事本程序：

```
>>>a=input('Please input a value:')
```

```
Please input a value:__import__('os').startfile(r'C:\Windows\\notepad.exe')
>>>eval(a)
```

下面的代码则会导致屏幕一闪，瞬间在当前文件夹中创建了一个子文件夹 testtest：

```
>>>eval("__import__('os').system('md testtest')")
```

因此，如果我们的程序中有使用内置函数 eval( )对用户输入的字符串求值的代码，一定要检查用户输入的字符串中是否有危险的字符串并对这些特殊的字符串进行必要的过滤，例如"__import__('os')."，否则就很容易引发很多安全问题。当然，也可以使用标准库 ast 提供的安全函数 literal_eval( )。

7.4.12 字符串对象的切片操作

切片也适用于字符串，但仅限于读取其中的元素，不支持字符串修改。

```
>>>'Explicit is better than implicit.'[:8]
'Explicit'
>>>'Explicit is better than implicit.'[9:23]
'is better than'
>>>path='C:\\Python35\\test.bmp'
>>>path[:-4] +'_new' +path[-4:]
'C:\\Python35\\test_new.bmp'
```

7.5 其他有关模块

7.5.1 textwrap 模块

前面介绍的 center( )、ljust( )等字符串方法支持一定的排版功能，除此之外，Python 标准库 textwrap 提供了更加友好的排版函数。

(1) wrap(text, width = 70)函数对一段文本进行自动换行，每一行不超过 width 个字符。

```
>>>import textwrap
>>>doc='''Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.'''
>>>import pprint
>>>pprint.pprint(textwrap.wrap(doc))                    #默认长度最大为 70
```




```
['Beautiful is better than ugly. Explicit is better than implicit.',
'Simple is better than complex. Complex is better than complicated.',
'Flat is better than nested. Sparse is better than dense. Readability',
"counts. Special cases aren't special enough to break the rules.",
'Although practicality beats purity.']
```

(2) fill(text, width=70)函数对一段文本进行排版和自动换行,等价于'\n'.join(wrap(text,…))。

```
>>>print(textwrap.fill(doc,width=20))        #按指定宽度进行排版
'Beautiful is better
than ugly. Explicit
is better than
implicit. Simple is
better than complex.
Complex is better
than complicated.
Flat is better than
nested. Sparse is
better than dense.
Readability counts.
Special cases aren't
special enough to
break the rules.
Although
practicality beats
purity.
>>>print(textwrap.fill(doc,width=80))        #按指定宽度进行排版
'Beautiful is better than ugly. Explicit is better than implicit. Simple is
better than complex. Complex is better than complicated. Flat is better than
nested. Sparse is better than dense. Readability counts. Special cases aren't
special enough to break the rules. Although practicality beats purity.
```

(3) shorten(text, width, **kwargs)函数截断文本以适应指定的宽度。该函数首先把文本中的所有连续空白字符替换(或折叠)为一个空白字符,如果能够适应指定的宽度就返回,否则就在文本尾部**丢弃足够多的单词**并替换为指定的占位符。

```
>>>from textwrap import shorten
>>>shorten('Hello      world!',width=15)          #宽度足以容纳所有字符
'Hello world!'
>>>shorten('Hello      world!',width=10)          #指定的宽度太小
'[…]'
>>>shorten('Hello      world!',width=11)
'Hello […]'
>>>shorten('Hello      world!',width=11,placeholder='.')  #指定占位符
```

```
'Hello.'
>>>shorten('Hello      world!',width=11,placeholder='…')  #使用不同的占位符
'Hello…'
>>>shorten('Hello      world!',width=5,placeholder='…')   #指定的宽度太小
'…'
```

(4) indent(text, prefix, predicate = None)函数对文本进行缩进并为所有非空行增加指定的前导字符或前缀,通过 predicate 可以更加灵活地控制为哪些行增加前导字符。

```
>>>from textwrap import indent
>>>example='''
hello
  world
  a

good'''
>>>print(indent(example,'+'* 4))                          #默认为所有非空行增加前缀
++++hello
++++  world
++++  a

++++good
>>>print(indent(example,'+'* 4,lambda line:True))         #为所有行增加前缀
++++
++++hello
++++  world
++++  a
++++
++++good
>>>print(indent(example,'+'* 4,lambda line:len(line)<3))          #只为长度小于 3 的行增加前缀
++++
hello
  world
  a
++++
good
```

(5) dedent(text)函数用来删除文本中每一行的所有**公共前导空白字符**。

```
>>>from textwrap import dedent
>>>example='''
    hello
      world
    good'''
>>>print(dedent(example))
hello
  world
```



```
good
```

(6) TextWrapper 类。

前面介绍的 wrap()、fill()、shorten()函数在内部都是先创建一个 TextWrapper 类的实例,然后再调用该实例的方法。如果需要频繁调用这几个函数的话,就会重复创建 TextWrapper 类的实例,严重影响效率,可以创建 TextWrapper 类的实例然后再调用该实例的方法。

```
>>>from textwrap import TextWrapper
>>>wrapper =TextWrapper(width =70,initial_indent =' +',placeholder ='…')
>>>print(wrapper.wrap('hello world' * 40))
[' +hello worldhello worldhello worldhello worldhello worldhello', 'worldhello worldhello worldhello
worldhello worldhello worldhello', 'worldhello worldhello worldhello worldhello worldhello worldhello
', ' worldhello worldhello worldhello worldhello worldhello worldhello ', ' worldhello worldhello
worldhello worldhello worldhello worldhello', 'worldhello worldhello worldhello worldhello worldhello
worldhello', 'worldhello worldhello worldhello worldhello world']
>>>print(wrapper.fill('hello world' * 40))
 +hello worldhello worldhello worldhello worldhello worldhello
worldhello worldhello worldhello worldhello worldhello worldhello
worldhello worldhello worldhello worldhello worldhello worldhello
worldhello worldhello worldhello worldhello worldhello worldhello
worldhello worldhello worldhello worldhello worldhello worldhello
worldhello worldhello worldhello worldhello worldhello worldhello
worldhello worldhello worldhello worldhello world
```

7.5.2　zlib 模块提供的压缩功能

Pytho 标准库 zlib 中提供的 compress()和 decompress()函数可以用于数据的压缩和解压缩,在压缩字符串之前需要先编码为字节串。

```
>>>import zlib
>>>x ='Python 程序设计系列图书,董付国编著,清华大学出版社'.encode()
>>>len(x)
72
>>>y =zlib.compress(x)
>>>len(y)                                    #对于重复度比较小的信息,压缩比小
83
>>>x =('Python 系列图书' * 3).encode()
>>>len(x)
54
>>>y =zlib.compress(x)                       #信息重复度越高,压缩比越大
>>>len(y)
30
>>>z =zlib.decompress(y)
>>>len(z)
```

```
54
>>>z.decode()
'Python 系列图书 Python 系列图书 Python 系列图书'
>>>x=['董付国'] *8
>>>y=str(x).encode()
>>>len(y)
104
>>>z=zlib.compress(y)                       #只能对字节串进行压缩
>>>len(z)
26
>>>zlib.decompress(z).decode()
"['董付国','董付国','董付国','董付国','董付国','董付国','董付国','董付国']"
```

7.6 字符串常量

Python 标准库 string 提供了英文字母大小写、数字字符、标点符号等常量，可以直接使用。下面的代码实现了随机密码生成功能。

```
>>>import string
>>>x=string.digits +string.ascii_letters +string.punctuation
                                            #可能的字符集
>>>x
'0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&\'()*+,-./:;<=>?@[\]^_`
{|}~'
>>>import random
>>>def generateStrongPwd(k):                #生成指定长度的随机密码字符串
    return ''.join((random.choice(x) for i in range(k)))
>>>generateStrongPwd(8)                     #8 位随机密码
'@<JnOR$i'
>>>generateStrongPwd(8)
'o.u:E+lv'
>>>generateStrongPwd(15)                    #15 位随机密码
'^0|O*7Gwi.u..e/'
```

7.7 可变字符串

在 Python 中，字符串属于不可变对象，不支持原地修改，如果需要修改其中的值，只能重新创建一个新的字符串对象。如果确实需要一个支持原地修改的 UNICODE 字符串对象，可以使用 io.StringIO 对象或 array 模块。

```
>>>from io import StringIO
>>>s="Hello world"
>>>sio=StringIO(s)                      #创建可变字符串对象
>>>sio
```

```
<_io.StringIO object at 0x0000000003096EE8>
>>>sio.tell()                          #返回当前位置
0
>>>sio.read()                          #从当前位置开始读取字符串
'Hello world'
>>>sio.getvalue()                      #返回可变字符串的全部内容
'Hello world'
>>>sio.tell()
11
>>>sio.seek(6)                         #重新定位当前位置
6
>>>sio.write('SDIBT')                  #从当前位置开始写入字符串,自动移动指针
5
>>>sio.read()                          #从当前位置开始读取字符串
''
>>>sio.getvalue()
'Hello SDIBT'
>>>sio.tell()
11
>>>s="Hello world"
>>>from array import array
>>>sa=array('u',s)                     #创建可变字符串对象
>>>print(sa)
array('u','Hello world')
>>>print(sa.tostring())                #查看可变字符串对象内容
b'H\x00e\x00l\x00l\x00o\x00 \x00w\x00o\x00r\x00l\x00d\x00'
>>>print(sa.tounicode())               #查看可变字符串对象内容
Hello world
>>>sa[0]='F'                           #修改指定位置上的字符
>>>print(sa)
array('u','Fello world')
>>>sa.insert(5,'w')                    #在指定位置插入字符
>>>print(sa)
array('u','Fellow world')
>>>sa.remove('l')                      #删除指定字符的首次出现
>>>print(sa)
array('u','Felow world')
>>>sa.remove('w')
>>>print(sa)
array('u','Felo world')
```

7.8　中英文分词

如果字符串中有连续的英文和中文,可以根据字符串的规律自己编写代码将其切分。

```
x='狗 dog猫 cat杯子 cup桌子 table你好'
c=[]
e=[]
t=''
for ch in x:                                    #先提取英文
    if 'a'<=ch<='z' or 'A'<=ch<='Z':
        t +=ch
    elif t:
        e.append(t)
        t=''
if t:
    e.append(t)
    t=''

for ch in x:                                    #再提取中文
    if 0x4e00<=ord(ch)<=0x9fa5:                 #基本汉字 UNICODE 编码范围
        t +=ch
    elif t:
        c.append(t)
        t=''
if t:
    c.append(t)
    t=''

print(c)
print(e)
```

Python 扩展库 jieba 和 snownlp 很好地支持了中文分词，可以使用 pip 命令进行安装。在自然语言处理领域经常需要对文字进行分词，分词的准确度直接影响了后续文本处理和挖掘算法的最终效果。

```
>>>import jieba                                  #导入 jieba 模块
>>>x='分词的准确度直接影响了后续文本处理和挖掘算法的最终效果。'
>>>jieba.cut(x)                                  #使用默认词库进行分词
<generator object Tokenizer.cut at 0x000000000342C990>
>>>list(_)
['分词','的','准确度','直接','影响','了','后续','文本处理','和','挖掘','算法','的','最终',
'效果','。']
>>>list(jieba.cut('纸杯'))
['纸杯']
>>>list(jieba.cut('花纸杯'))
['花','纸杯']
>>>jieba.add_word('花纸杯')                      #增加词条
>>>list(jieba.cut('花纸杯'))                     #使用新题库进行分词
['花纸杯']
```

```
>>>import snownlp                          #导入 snownlp 模块
>>>snownlp.SnowNLP('学而时习之,不亦说乎').words
['学而','时习','之',',','不亦','说乎']
>>>snownlp.SnowNLP(x).words
['分词','的','准确度','直接','影响','了','后续','文本','处理','和','挖掘','算法','的','最终','效果','。']
```

7.9　汉字到拼音的转换

Python 扩展库 pypinyin 支持汉字到拼音的转换,并且可以和分词扩展库配合使用。

```
>>>from pypinyin import lazy_pinyin,pinyin
>>>lazy_pinyin('董付国')                   #返回拼音
['dong','fu','guo']
>>>lazy_pinyin('董付国',1)                 #带声调的拼音
['dǒng','fù','guó']
>>>lazy_pinyin('董付国',2)                 #另一种拼音形式,数字表示前面字母的声调
['do3ng','fu4','guo2']
>>>lazy_pinyin('董付国',3)                 #只返回拼音首字母
['d','f','g']
>>>lazy_pinyin('重要',1)                   #能够根据词组智能识别多音字
['zhòng','yào']
>>>lazy_pinyin('重阳',1)
['chóng','yáng']
>>>pinyin('重阳')                          #返回拼音
[['chóng'],['yáng']]
>>>pinyin('重阳节',heteronym=True)         #返回多音字的所有读音
[['zhòng','chóng','tóng'],['yáng'],['jié','jiē']]
>>>import jieba                            #其实不需要导入 jieba,这里只是说明已安装
>>>x='中英文混合 test123'
>>>lazy_pinyin(x)                          #自动调用已安装的 jieba 扩展库分词功能
['zhong','ying','wen','hun','he','test123']
>>>lazy_pinyin(jieba.cut(x))
['zhong','ying','wen','hun','he','test123']
>>>x='山东烟台的大樱桃真好吃啊'
>>>sorted(x,key=lambda ch: lazy_pinyin(ch))
                                                   #按拼音对汉字进行排序
['啊','吃','大','的','东','好','山','台','桃','烟','樱','真']
```

7.10　精彩案例赏析

示例 7-1　编写函数实现字符串加密和解密,循环使用指定密钥,采用简单的异或算法。

```
def crypt(source,key):
    from itertools import cycle
    result=''
    temp=cycle(key)
    for ch in source:
        result=result +chr(ord(ch) ^ ord(next(temp)))
    return result

source='Shandong Institute of Business and Technology'
key='Dong Fuguo'

print('Before Encrypted:' +source)
encrypted=crypt(source,key)
print('After Encrypted:' +encrypted)
decrypted=crypt(encrypted,key)
print('After Decrypted:' +decrypted)
```

输出结果如图 7-2 所示。

```
Before Encrypted:Shandong Institute of Business and Technology
After Encrypted:┤●☼     D)← U&*→♬T3  ┐U ˝O,↨S/↩┐−d♬ └ ↨┼┘  +└   Y
After Decrypted:Shandong Institute of Business and Technology
```

图 7-2 字符串加密与解密结果

示例 7-2 编写程序,生成大量随机信息。

本例代码演示了如何使用 Python 标准库 random 来生成随机数据,这在需要获取大量数据来测试或演示软件功能的时候非常有用,不仅能真实展示软件的功能或算法,还可以避免泄露真实数据引起不必要的争议。

```
from random import choice,randint
import string
import codecs

#常用汉字 Unicode 编码表(部分),完整列表详见配套源代码
StringBase='\u7684\u4e00\u4e86\u662f\u6211\u4e0d\u5728\u4eba'

def getEmail():
    #常见域名后缀,可以随意扩展该列表
    suffix=['.com','.org','.net','.cn']
    characters=string.ascii_letters+string.digits+'_'
    username=''.join((choice(characters) for i in range(randint(6,12))))
    domain=''.join((choice(characters) for i in range(randint(3,6))))
    return username+'@'+domain+choice(suffix)

def getTelNo():
    return ''.join((str(randint(0,9)) for i in range(11)))
```

```
def getNameOrAddress(flag):
    '''flag=1 表示返回随机姓名,flag=0 表示返回随机地址'''
    if flag==1:
        #大部分中国人姓名为 2~5 个汉字
        rangestart,rangeend=2,5
    elif flag==0:
        #假设地址在 10~30 个汉字之间
        rangestart,rangeend=10,30
    result=''.join((choice(StringBase) for i in range(randint(rangestart,rangeend))))
    return result

def getSex():
    return choice(('男','女'))

def getAge():
    return str(randint(18,100))

def main(filename):
    with codecs.open(filename,'w','utf-8') as fp:
        fp.write('Name,Sex,Age,TelNO,Address,Email\n')
        #随机生成 200 个人的信息
        for i in range(200):
            name=getNameOrAddress(1)
            sex=getSex()
            age=getAge()
            tel=getTelNo()
            address=getNameOrAddress(0)
            email=getEmail()
            line=','.join([name,sex,age,tel,address,email]) +'\n'
            fp.write(line)

def output(filename):
    with codecs.open(filename,'r','utf-8') as fp:
        for line in fp:
            line=line.split(',')
            for i in line:
                print(i,end=',')
            print()

if __name__=='__main__':
    filename='information.txt'
    main(filename)
    output(filename)
```

示例 7-3　检查并判断密码字符串的安全强度。

```
import string

def check(pwd):
    #密码必须至少包含 6 个字符
    if not isinstance(pwd,str) or len(pwd)<6:
        return 'not suitable for password'

    #密码强度等级与包含字符种类的对应关系
    d={1:'weak',2:'below middle',3:'above middle',4:'strong'}
    #分别用来标记 pwd 是否含有数字、小写字母、大写字母和指定的标点符号
    r=[False]*4

    for ch in pwd:
        #是否包含数字
        if not r[0] and ch in string.digits:
            r[0]=True
        #是否包含小写字母
        elif not r[1] and ch in string.ascii_lowercase:
            r[1]=True
        #是否包含大写字母
        elif not r[2] and ch in string.ascii_uppercase:
            r[2]=True
        #是否包含指定的标点符号
        elif not r[3] and ch in ',.!;?<>':
            r[3]=True
    #统计包含的字符种类,返回密码强度
    return d.get(r.count(True),'error')

print(check('a2Cd,'))
```

第 8 章



文本处理(二):正则表达式

正则表达式是字符串处理的有力工具,正则表达式使用预定义的模式去匹配一类具有共同特征的字符串,可以快速、准确地完成复杂的查找、替换等处理要求,比字符串自身提供的方法提供了更强大的处理功能。例如,使用字符串对象的 split()方法只能指定一个分隔符,而使用正则表达式可以很方便地指定多个符号作为分隔符;使用字符串对象的 split()并指定分隔符时,很难处理分隔符连续多次出现的情况,而正则表达式让这一切都变得非常轻松。

8.1 正则表达式语法

8.1.1 正则表达式基本语法

正则表达式由元字符及其不同组合来构成,通过巧妙地构造正则表达式可以匹配任意字符串,完成查找、替换等复杂的字符串处理任务。常用的正则表达式元字符如表 8-1 所示。

表 8-1 正则表达式常用元字符

元字符	功能说明
.	匹配除换行符以外的任意单个字符
*	匹配位于 * 之前的字符或子模式的 0 次或多次出现
+	匹配位于 + 之前的字符或子模式的 1 次或多次出现
-	在[ ]之内用来表示范围
|	匹配位于|之前或之后的字符
^	匹配以^后面的字符或模式开头的字符串
$	匹配以 $ 前面的字符或模式结束的字符串
?	匹配位于“?”之前的 0 个或 1 个字符或子模式,即问号之前的字符或子模式是可选的。紧随任何其他限定符(*、+、?、{n}、{n,}、{n,m})之后时,表示“非贪心”匹配模式。“非贪心”模式匹配尽可能短的字符串,而默认的“贪心”模式匹配尽可能长的字符串。例如,在字符串"oooo"中,“o+?”只匹配单个 o,而 o+ 匹配所有 o

续表

元字符	功 能 说 明
\	表示位于\之后的为转义字符
\num	此处的 num 是一个正整数,表示前面字符或子模式的编号。例如,“(.)\1”匹配两个连续的相同字符
\f	匹配一个换页符
\n	匹配一个换行符
\r	匹配一个回车符
\b	匹配单词头或单词尾
\B	与\b 含义相反
\d	匹配任何数字,相当于[0-9]
\D	与\d 含义相反,相当于[^0-9]
\s	匹配任何空白字符,包括空格、制表符、换页符,与[\f\n\r\t\v]等效
\S	与\s 含义相反
\w	匹配任何字母、数字以及下画线,相当于[a-zA-Z0-9_]
\W	与\w 含义相反,与[^A-Za-z0-9_]等效
( )	将位于( )内的内容作为一个整体来对待
{ }	按{ }中指定的次数进行匹配,例如,{3,8}表示前面的字符或模式至少重复 3 次而最多重复 8 次
[ ]	匹配位于[ ]中的任意一个字符
[^xyz]	^放在[ ]内表示反向字符集,匹配除 x、y、z 之外的任何字符
[a-z]	字符范围,匹配指定范围内的任何字符
[^a-z]	反向范围字符,匹配除小写英文字母之外的任何字符

如果以\开头的元字符与转义字符相同,则需要使用\\,或者使用原始字符串。在字符串前加上字符 r 或 R 之后表示原始字符串,字符串中任意字符都不再进行转义。原始字符串可以减少用户的输入,主要用于正则表达式和文件路径字符串的情况,但如果字符串以一个斜线\结束,则需要多写一个斜线,即以\\结束。

8.1.2 正则表达式扩展语法

正则表达式使用圆括号“( )”表示一个子模式,圆括号内的内容作为一个整体对待,例如,'(red)+'可以匹配'redred'、'redredred'等一个或多个重复'red'的情况。使用子模式扩展语法可以实现更加复杂的字符串处理功能,常用的扩展语法如表 8-2 所示。



表 8-2　常用子模式扩展语法

语　法	功 能 说 明
(?P<groupname>)	为子模式命名
(?iLmsux)	设置匹配标志,可以是几个字母的组合,每个字母含义与编译标志相同
(?:…)	匹配但不捕获该匹配的子表达式
(?P=groupname)	表示在此之前的命名为 groupname 的子模式
(?#…)	表示注释
(?<=…)	用于正则表达式之前,如果<=后的内容在字符串中出现则匹配,但不返回<=之后的内容
(?=…)	用于正则表达式之后,如果=后的内容在字符串中出现则匹配,但不返回=之后的内容
(?<!…)	用于正则表达式之前,如果<!后的内容在字符串中不出现则匹配,但不返回<!之后的内容
(?!…)	用于正则表达式之后,如果!后的内容在字符串中不出现则匹配,但不返回!之后的内容

8.1.3 正则表达式锦集

正则表达式语法博大精深,很难一下子全都记住,建议在了解基本语法的基础上,记住一些常用的写法,然后在实际应用中不断深入。

(1) 最简单的正则表达式是普通字符串,只能匹配自身。

(2) '[pjc]ython'可以匹配'python'、'jython'、'cython'。

(3) '[a-zA-Z0-9]'可以匹配一个任意大小写字母或数字。

(4) '[^abc]'可以匹配一个任意除'a'、'b'、'c'之外的字符。

(5) 'python|perl'或'p(ython|erl)'都可以匹配'python'或'perl'。

(6) r'(http://)?(www\.)?python\.org'只能匹配'http://www.python.org'、'http://python.org'、'www.python.org'和'python.org'。

(7) '^http'只能匹配所有以'http'开头的字符串。

(8) (pattern)*:允许模式重复 0 次或多次。

(9) (pattern)+:允许模式重复一次或多次。

(10) (pattern){m,n}:允许模式重复 m~n 次,注意逗号后面不要有空格。

(11) '(a|b)*c':匹配多个(包含 0 个)a 或 b,后面紧跟一个字母 c。

(12) 'ab{1,}':等价于'ab+',匹配以字母 a 开头后面紧跟一个或多个字母 b 的字符串。

(13) '^[a-zA-Z]{1}([a-zA-Z0-9._]){4,19}$':匹配长度为 5~20 的字符串,必须以字母开头并且后面可带数字、字母、“_”“.”的字符串。

(14) '^(\w){6,20}$':匹配长度为 6~20 的字符串,可以包含字母、数字、下画线。

(15) '^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}$':检查给定字符串是否为合法 IP

地址。

(16) '^(13[4-9]\d{8})|(15[01289]\d{8})$'：检查给定字符串是否为移动手机号码。

(17) '^[a-zA-Z]+$'：检查给定字符串是否只包含英文大小写字母。

(18) '^\w+@(\w+\.)+\w+$'：检查给定字符串是否为合法电子邮件地址。

(19) '^(\-)?\d+(\.\d{1,2})?$'：检查给定字符串是否为最多带有 2 位小数的正数或负数。

(20) '[\u4e00-\u9fa5]'：匹配给定字符串中的常用汉字。

(21) '^\d{18}|\d{15}$'：检查给定字符串是否为合法身份证格式。

(22) '\d{4}-\d{1,2}-\d{1,2}'：匹配指定格式的日期，例如 2017-3-30。

(23) '^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[,._]).{8,}$'：检查给定字符串是否为强密码，必须同时包含英语字母大写字母、英文小写字母、数字或特殊符号（如英文逗号、英文句号、下画线），并且长度必须至少 8 位。

(24) "(?!.*[\'\"\/;=%?]).+"：如果给定字符串中包含'、"、/、;、=、% 和"?"则匹配失败，关于子模式语法请参考表 5-4。

(25) '(.)\\1+'：匹配任意字符或模式的一次或多次重复出现。

(26) '((?P<f>\b\w+\b)\s+(?P=f))'：匹配连续出现两次的单词。

(27) '((?P<f>.)(?P=f)(?P<g>.)(?P=g))'：匹配 AABB 形式的成语或字母组合。

使用时要注意的是，正则表达式只是进行形式上的检查，并不保证内容一定正确。例如上面的例子中，正则表达式'^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}$'可以检查字符串是否为 IP 地址，字符串'888.888.888.888'这样的也能通过检查，但实际上并不是有效的 IP 地址。同样的道理，正则表达式'^\d{18}|\d{15}$'也只负责检查字符串是否为 18 位或 15 位数字，并不保证一定是合法的身份证号。

8.2 直接使用正则表达式模块 re 处理字符串

Python 标准库 re 提供了正则表达式操作所需要的功能，既可以直接使用 re 模块中的方法（见表 8-3）处理字符串，也可以把模式编译成正则表达式对象再使用（见 8.3 节）。

表 8-3 re 模块中的方法

方 法	功 能 说 明
compile(pattern[, flags])	创建模式对象
escape(string)	将字符串中所有特殊正则表达式字符转义
findall(pattern, string[, flags])	列出字符串中模式的所有匹配项
finditer(pattern, string, flags=0)	返回包含所有匹配项的迭代对象，其中每个匹配项都是 match 对象

续表

方　　法	功 能 说 明
fullmatch(pattern, string, flags =0)	尝试把模式作用于整个字符串,返回 match 对象或 None
match(pattern, string[, flags])	从字符串的开始处匹配模式,返回 match 对象或 None
purge()	清空正则表达式缓存
search(pattern, string[, flags])	在整个字符串中寻找模式,返回 match 对象或 None
split(pattern, string[, maxsplit =0])	根据模式匹配项分隔字符串
sub(pat, repl, string[, count =0])	将字符串中所有 pat 的匹配项用 repl 替换,返回新字符串,repl 可以是字符串或返回字符串的可调用对象,该可调用对象作用于每个匹配的 match 对象
subn(pat, repl, string[, count =0])	将字符串中所有 pat 的匹配项用 repl 替换,返回包含新字符串和替换次数的二元组,repl 可以是字符串或返回字符串的可调用对象,该可调用对象作用于每个匹配的 match 对象

其中函数参数 flags 的值可以是 re.I(注意是大写字母 I,不是数字 1,表示忽略大小写)、re.L(支持本地字符集的字符)、re.M(多行匹配模式)、re.S(使元字符“.”匹配任意字符,包括换行符)、re.U(匹配 Unicode 字符)、re.X(忽略模式中的空格,并可以使用#注释)的不同组合(使用|进行组合)。

下面的代码演示了直接使用 re 模块中的方法和正则表达式处理字符串的用法,其中 match()函数用于在字符串开始位置进行匹配,而 search()函数用于在整个字符串中进行匹配,这两个函数如果匹配成功则返回 match 对象,否则返回 None。

```
>>>import re                                    #导入 re 模块
>>>text ='alpha. beta···gamma delta'            #测试用的字符串
>>>re.split('[\. ] +',text)                     #使用指定字符作为分隔符进行分隔
['alpha','beta','gamma','delta']
>>>re.split('[\. ] +',text,maxsplit =2)         #最多分隔 2 次
['alpha','beta','gamma delta']
>>>re.split('[\. ] +',text,maxsplit =1)         #最多分隔 1 次
['alpha','beta···gamma delta']
>>>pat ='[a -zA -Z] +'
>>>re.findall(pat,text)                         #查找所有单词
['alpha','beta','gamma','delta']
>>>pat ='{name}'
>>>text ='Dear {name}···'
>>>re.sub(pat,'Mr.Dong',text)                   #字符串替换
'Dear Mr.Dong···'
>>>s ='a s d'
>>>re.sub('a|s|d','good',s)                     #字符串替换
'good good good'
>>>s ="It's a very good good idea"
```

```
>>>re.sub(r'(\b\w+) \1',r'\1',s)              #处理连续的重复单词
"It's a very good idea"
>>>re.sub('a',lambda x:x.group(0).upper(),'aaa abc abde')
                                              #repl 为可调用对象
'AAA Abc Abde'
>>>re.sub('[a-z]',lambda x:x.group(0).upper(),'aaa abc abde')
'AAA ABC ABDE'
>>>re.sub('[a-zA-z]',lambda x:chr(ord(x.group(0))^32),'aaa aBc abde')
                                              #英文字母大小写互换
'AAA AbC ABDE'
>>>re.subn('a','dfg','aaa abc abde')          #返回新字符串和替换次数
('dfgdfgdfg dfgbc dfgbde',5)
>>>re.sub('a','dfg','aaa abc abde')
'dfgdfgdfg dfgbc dfgbde'
>>>re.escape('http://www.python.org')         #字符串转义
'http\\:\\/\\/www\\.python\\.org'
>>>print(re.match('done|quit','done'))        #匹配成功,返回 match 对象
<_sre.SRE_Match object at 0x00B121A8>
>>>print(re.match('done|quit','done!'))       #匹配成功
<_sre.SRE_Match object at 0x00B121A8>
>>>print(re.match('done|quit','doe!'))        #匹配不成功,返回空值 None
None
>>>print(re.match('done|quit','d!one!'))      #匹配不成功
None
>>>print(re.search('done|quit','d!one!done')) #匹配成功
<_sre.SRE_Match object at 0x0000000002D03D98>
```

下面的代码使用不同的方法删除字符串中多余的空格,如果遇到连续多个空格则只保留一个,同时删除字符串两侧的所有空白字符。

```
>>>import re
>>>s='aaa       bb      c d e   fff    '
>>>' '.join(s.split())                         #直接使用字符串对象的方法
'aaa bb c d e fff'
>>>' '.join(re.split('[\s]+',s.strip()))       #同时使用 re 模块中的函数和字符串对象的方法
'aaa bb c d e fff'
>>>' '.join(re.split('\s+',s.strip()))         #与上一行代码等价
'aaa bb c d e fff'
>>>re.sub('\s+',' ',s.strip())                 #直接使用 re 模块的字符串替换方法
'aaa bb c d e fff'
```

下面的代码使用几种不同的方法来删除字符串中指定的内容:

```
>>>email="tony@tiremove_thisger.net"
```



```
>>>m=re.search("remove_this",email)          #使用 search()方法返回的 match 对象
>>>email[:m.start()] +email[m.end():]         #字符串切片
'tony@tiger.net'
>>>re.sub('remove_this','',email)             #直接使用 re 模块的 sub()方法
'tony@tiger.net'
>>>email.replace('remove_this','')            #直接使用字符串替换方法
'tony@tiger.net'
```

下面的代码使用以\开头的元字符来实现字符串的特定搜索。

```
>>>import re
>>>example='Beautiful is better than ugly.'
>>>re.findall('\\bb.+?\\b',example)           #以字母 b 开头的完整单词
                                              #此处问号"?"表示非贪心模式
['better']
>>>re.findall('\\bb.+\\b',example)            #贪心模式的匹配结果
['better than ugly']
>>>re.findall('\\bb\w*\\b',example)
['better']
>>>re.findall('\\Bh.+?\\b',example)           #不以 h 开头且含有 h 字母的单词剩余部分
['han']
>>>re.findall('\\b\w.+?\\b',example)          #所有单词
['Beautiful','is','better','than','ugly']
>>>re.findall('\w+',example)                  #所有单词
['Beautiful','is','better','than','ugly']
>>>re.findall(r'\b\w.+?\b',example)           #使用原始字符串
['Beautiful','is','better','than','ugly']
>>>re.split('\s',example)                     #使用任何空白字符分隔字符串
['Beautiful','is','better','than','ugly.']
>>>re.findall('\d+\.\d+\.\d+','Python 2.7.13')
                                              #查找并返回 x.x.x形式的数字
['2.7.13']
>>>re.findall('\d+\.\d+\.\d+','Python 2.7.13,Python 3.6.0')
['2.7.13','3.6.0']
>>>s='<html><head>This is head.</head><body>This is body.</body></html>'
>>>pattern=r'<html><head>(.+)</head><body>(.+)</body></html>'
>>>result=re.search(pattern,s)
>>>result.group(1)                            #第一个子模式
'This is head.'
>>>result.group(2)                            #第二个子模式
'This is body.'
```

8.3　使用正则表达式对象处理字符串

虽然直接使用 re 模块也可以使用正则表达式处理字符串,但是正则表达式对象提供

了更多的功能。使用编译后的正则表达式对象不仅可以提高字符串的处理速度，还提供了更加强大的字符串处理功能。首先使用 re 模块的 compile()方法将正则表达式编译成正则表达式对象，然后再使用正则表达式对象提供的方法进行字符串处理。

1. match()、search()、findall()

正则表达式对象的 match(string[, pos[, endpos]])方法在字符串开头或指定位置进行搜索，模式必须出现在字符串开头或指定位置；search(string[, pos[, endpos]])方法在整个字符串或指定范围中进行搜索；findall(string[, pos[, endpos]])方法字符串中查找所有符合正则表达式的字符串并以列表形式返回。

```
>>>import re
>>>example='ShanDong Institute of Business and Technology'
>>>pattern=re.compile(r'\bB\w+\b')              #查找以 B 开头的单词
>>>pattern.findall(example)                     #使用正则表达式对象的 findall()方法
['Business']
>>>pattern=re.compile(r'\w+g\b')                #查找以字母 g 结尾的单词
>>>pattern.findall(example)
['ShanDong']
>>>pattern=re.compile(r'\b[a-zA-Z]{3}\b')       #查找 3 个字母长的单词
>>>pattern.findall(example)
['and']
>>>pattern.match(example)                       #从字符串开头开始匹配,失败返回空值
>>>pattern.search(example)                      #在整个字符串中搜索,成功
<_sre.SRE_Match object; span=(31,34),match='and'>
>>>pattern=re.compile(r'\b\w*a\w*\b')           #查找所有含有字母 a 的单词
>>>pattern.findall(example)
['ShanDong','and']
>>>text="He was carefully disguised but captured quickly by police."
>>>re.findall(r"\w+ly",text)                    #查找所有以字母组合 ly 结尾的单词
['carefully','quickly']
```

2. sub()、subn()

正则表达式对象的 sub(repl, string[, count=0])和 subn(repl, string[, count=0])方法用来实现字符串替换功能，其中参数 repl 可以为字符串或返回字符串的可调用对象。

```
>>>example='''Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.'''
```

```
>>>pattern=re.compile(r'\bb\w*\b',re.I)        #匹配以 b 或 B 开头的单词
>>>print(pattern.sub('*',example))             #将符合条件的单词替换为*
* is* than ugly.
Explicit is* than implicit.
Simple is* than complex.
Complex is* than complicated.
Flat is* than nested.
Sparse is* than dense.
Readability counts.
>>>print(pattern.sub(lambda x: x.group(0).upper(),example))
                                               #把所有匹配项都改为大写
BEAUTIFUL is BETTER than ugly.
Explicit is BETTER than implicit.
Simple is BETTER than complex.
Complex is BETTER than complicated.
Flat is BETTER than nested.
Sparse is BETTER than dense.
Readability counts.
>>>print(pattern.sub('*',example,1))           #只替换一次
* is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
>>>pattern=re.compile(r'\bb\w*\b')             #匹配以字母 b 开头的单词
>>>print(pattern.sub('*',example,1))           #将符合条件的单词替换为*
                                               #只替换一次
Beautiful is* than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
```

3. split()

正则表达式对象的 split(string[, maxsplit=0])方法用来实现字符串分隔。

```
>>>example=r'one,two,three.four/five\six?seven[eight]nine|ten'
>>>pattern=re.compile(r'[,./\\?[]\|]')         #指定多个可能的分隔符
>>>pattern.split(example)
['one','two','three','four','five','six','seven','eight','nine','ten']
```

```
>>>example=r'one1two2three3four4five5six6seven7eight8nine9ten'
>>>pattern=re.compile(r'\d+')                    #使用数字作为分隔符
>>>pattern.split(example)
['one','two','three','four','five','six','seven','eight','nine','ten']
>>>example=r'one two    three   four,five.six.seven,eight,nine9ten'
>>>pattern=re.compile(r'[\s,.\d]+')              #允许分隔符重复
>>>pattern.split(example)
['one','two','three','four','five','six','seven','eight','nine','ten']
```

8.4 match 对象

正则表达式模块或正则表达式对象的 match()方法和 search()方法匹配成功后都会返回 match 对象。match 对象的主要方法有 group()(返回匹配的一个或多个子模式内容)、groups()(返回一个包含匹配的所有子模式内容的元组)、groupdict()(返回包含匹配的所有命名子模式内容的字典)、start()(返回指定子模式内容的起始位置)、end()(返回指定子模式内容的结束位置的前一个位置)、span()(返回一个包含指定子模式内容起始位置和结束位置前一个位置的元组)等。

下面的代码演示了 match 对象的 group()、groups()与 groupdict()以及其他方法的用法:

```
>>>m=re.match(r"(\w+) (\w+)","Isaac Newton,physicist")
>>>m.group(0)                                    #返回整个模式内容
'Isaac Newton'
>>>m.group(1)                                    #返回第一个子模式内容
'Isaac'
>>>m.group(2)                                    #返回第二个子模式内容
'Newton'
>>>m.group(1,2)                                  #返回指定的多个子模式内容
('Isaac','Newton')
```

下面的代码演示了子模式扩展语法的用法:

```
>>>m=re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)","Malcolm Reynolds")
>>>m.group('first_name')                         #使用命名的子模式
'Malcolm'
>>>m.group('last_name')
'Reynolds'
>>>m=re.match(r"(\d+)\.(\d+)","24.1632")
>>>m.groups()                                    #返回所有匹配的子模式(不包括第0个)
('24','1632')
>>>m=re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)","Malcolm Reynolds")
>>>m.groupdict()                                 #以字典形式返回匹配的结果
{'first_name': 'Malcolm','last_name': 'Reynolds'}
>>>exampleString='''There should be one--and preferably only one --obvious way to do it.
```

```
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than right now.'''
>>>pattern=re.compile(r'(?<=\w\s)never(?=\s\w)')
                                                    #查找不在句子开头和结尾的 never
>>>matchResult=pattern.search(exampleString)
>>>matchResult.span()
(172,177)
>>>pattern=re.compile(r'(?<=\w\s)never')
                                                    #查找位于句子末尾的单词
>>>matchResult=pattern.search(exampleString)
>>>matchResult.span()
(156,161)
>>>pattern=re.compile(r'(?:is\s)better(\sthan)')
                                                    #查找前面是 is 的 better than 组合
>>>matchResult=pattern.search(exampleString)
>>>matchResult.span()
(141,155)
>>>matchResult.group(0)                             #组 0 表示整个模式
'is better than'
>>>matchResult.group(1)
' than'
>>>pattern=re.compile(r'\b(?i)n\w+\b')
                                                    #查找以 n 或 N 字母开头的所有单词
>>>index=0
>>>while True:
    matchResult=pattern.search(exampleString,index)
    if not matchResult:
        break
    print(matchResult.group(0),':',matchResult.span(0))
    index=matchResult.end(0)
not : (92,95)
Now : (137,140)
never : (156,161)
never : (172,177)
now : (205,208)
>>>pattern=re.compile(r'(?<!not\s)be\b')           #查找前面没有单词 not 的单词 be
>>>index=0
>>>while True:
    matchResult=pattern.search(exampleString,index)
    if not matchResult:
        break
    print(matchResult.group(0),':',matchResult.span(0))
    index=matchResult.end(0)
```

```
be : (13,15)
>>>exampleString[13:20]                        #验证一下结果是否正确
'be one -'
>>>pattern =re.compile(r'(\b\w* (?P<f>\w+)(?P=f)\w* \b)')
                                               #匹配有连续相同字母的单词
>>>index =0
>>>while True:
    matchResult =pattern.search(exampleString,index)
    if not matchResult:
        break
    print(matchResult.group(0),':',matchResult.group(2))
    index =matchResult.end(0) +1
unless : s
better : t
better : t
>>>s ='aabc abcd abbcd abccd abcdd'
>>>p =re.compile(r'(\b\w* (?P<f>\w+)(?P=f)\w* \b)')
>>>p.findall(s)
[('aabc','a'),('abbcd','b'),('abccd','c'),('abcdd','d')]
```

8.5 精彩案例赏析

示例 8-1 使用正则表达式提取字符串中的电话号码。

```
import re

telNumber ='''Suppose my Phone No. is 0535 -1234567,
              yours is 010 -12345678,
              his is 025 -87654321.'''
pattern =re.compile(r'(\d{3,4})-(\d{7,8})')     #注意,逗号后面不能有空格
index =0
while True:
    matchResult =pattern.search(telNumber,index)  #从指定位置开始匹配
    if not matchResult:
        break
    print('-'* 30)
    print('Success:')
    for i in range(3):
        print('Searched content:',matchResult.group(i),\
          ' Start from:',matchResult.start(i),'End at:',matchResult.end(i),\
          ' Its span is:',matchResult.span(i))
    index =matchResult.end(2)                   #指定下次匹配的开始位置
```

示例 8-2 使用正则表达式提取 Python 程序中的类名、函数名以及变量名等标识符。



将下面的代码保存为 FindIdentifiersFromPyFile. py，在命令提示符环境中使用命令"Python FindIdentifiersFromPyFile. py 目标文件名"查找并输出目标文件中的标识符。

```
import re
import os
import sys

classes={}
functions=[]
variables={'normal':{},'parameter':{},'infor':{}}

'''This is a test string:
atest,btest=3,5
to verify that variables in comments will be ignored by this algorithm
'''

def _identifyClassNames(index,line):
    '''parameter index is the line number of line,
     parameter line is a line of code of the file to check'''
    pattern=re.compile(r'(?<=class\s)\w+(?=.*?:)')
    matchResult=pattern.search(line)
    if not matchResult:
        return
    className=matchResult.group(0)
    classes[className]=classes.get(className,[])
    classes[className].append(index)

def _identifyFunctionNames(index,line):
    pattern=re.compile(r'(?<=def\s)(\w+)\((.*?)\)(?=:)')
    matchResult=pattern.search(line)
    if not matchResult:
        return
    functionName=matchResult.group(1)
    functions.append((functionName,index))
    parameters=matchResult.group(2).split(r',')
    if parameters[0]=='':
        return
    for v in parameters:
        variables['parameter'][v]=variables['parameter'].get(v,[])
        variables['parameter'][v].append(index)

def _identifyVariableNames(index,line):
    #find normal variables,including the case: a,b=3,5
    pattern=re.compile(r'\b(.*?)(?=\s=)')
```

```
        matchResult=pattern.search(line)
        if matchResult:
            vs=matchResult.group(1).split(r',')
            for v in vs:
                #consider the case 'if variable==value'
                if 'if ' in v:
                    v=v.split()[1]
                #consider the case: 'a[3]=3'
                if '[' in v:
                    v=v[0:v.index('[')]
                variables['normal'][v]=variables['normal'].get(v,[])
                variables['normal'][v].append(index)
        #find the variables in for statements
        pattern=re.compile(r'(?<=for\s)(.*?)(?=\sin)')
        matchResult=pattern.search(line)
        if matchResult:
            vs=matchResult.group(1).split(r',')
            for v in vs:
                variables['infor'][v]=variables['infor'].get(v,[])
                variables['infor'][v].append(index)

def output():
    print('='*30)
    print('The class names and their line numbers are:')
    for key,value in classes.items():
        print(key,':',value)
    print('='*30)
    print('The function names and their line numbers are:')
    for i in functions:
        print(i[0],':',i[1])
    print('='*30)
    print('The normal variable names and their line numbers are:')
    for key,value in variables['normal'].items():
        print(key,':',value)
    print('-'*20)
    print('The parameter names and their line numbers in functions are:')
    for key,value in variables['parameter'].items():
        print(key,':',value)
    print('-'*20)
    print('The variable names and their line numbers in for statements are:')
    for key,value in variables['infor'].items():
        print(key,':',value)

#suppose the lines of comments less than 50
```

```
def comments(index):
    for i in range(50):
        line =allLines[index +i].strip()
        if line.endswith('"""') or line.endswith("'''"):
            return i +1

if __name__=='__main__':
    fileName =sys.argv[1]                         #命令行参数
    if not os.path.isfile(fileName):
        print('Your input is not a file.')
        sys.exit(0)                               #退出当前程序
    if not fileName.endswith('.py'):
        print('Sorry. I can only check Python source file.')
        sys.exit(0)
    allLines =[]
    with open(fileName,'r') as fp:
        allLines =fp.readlines()
    index =0
    totalLen =len(allLines)
    while index <totalLen:
        line =allLines[index]
        #strip the blank characters at both end of line
        line =line.strip()
        #ignore the comments starting with '#
        if line.startswith('#'):
            index +=1
            continue
        #ignore the comments between ''' or """
        if line.startswith('"""') or line.startswith("'''"):
            index +=comments(index)
            continue
        #identify identifiers
        _identifyClassNames(index +1,line)
        _identifyFunctionNames(index +1,line)
        _identifyVariableNames(index +1,line)
        index +=1
    output()
```

示例 8-3　使用正则表达式检查 Python 程序的代码风格是否符合规范。

本例代码主要检查 Python 程序的一些基本规范,例如,运算符两侧是否有空格,是否每次只导入一个模块,在不同的功能模块之间是否有空行,注释是否足够多,等等。本例代码用法与示例 8-2 中的代码用法一样。

```
import sys
```

```
import re

def checkFormats(lines,desFileName):
    fp =open(desFileName,'w')
    for i,line in enumerate(lines):
        print('=' * 30)
        print('Line:',i +1)
        if line.strip().startswith('#'):
            print(' ' * 10 +'Comments.Pass.')
            fp.write(line)
            continue
        flag =True
        #check operator symbols
        symbols =[',','+','-','*','/','//','**','>>','<<','+=','-=','*=','/=']
        temp_line =line
        for symbol in symbols:
            pattern =re.compile(r'\s*' +re.escape(symbol) +r'\s*')
            temp_line =pattern.split(temp_line)
            sep =' ' +symbol +' '
            temp_line =sep.join(temp_line)
        if line !=temp_line:
            flag =False
            print(' ' * 10 +'You may miss some blank spaces in this line.')
        #check import statement
        if line.strip().startswith('import'):
            if ',' in line:
                flag =False
                print(' ' * 10 +"You'd better import one module at a time.")
                temp_line =line.strip()
                modules =temp_line[temp_line.index(' ') +1:]
                modules =modules.strip()
                pattern =re.compile(r'\s*,\s*')
                modules =pattern.split(modules)
                temp_line =''
                for module in modules:
                    temp_line +=line[:line.index('import')] +'import ' +module +'\n'
                line =temp_line
            pri_line =lines[i -1].strip()
            if pri_line and (not pri_line.startswith('import')) and \
              (not pri_line.startswith('#')):
                flag =False
                print(' ' * 10 +'You should add a blank line before this line.')
                line ='\n' +line
            after_line =lines[i +1].strip()
```



```
            if after_line and (not after_line.startswith('import')):
                flag=False
                print(' ' * 10 +'You should add a blank line after this line.')
                line=line+'\n'
        #check if there is a blank line before new funtional code block
        #including the class/function definition
        if line.strip() and not line.startswith(' ') and i>0:
            pri_line=lines[i-1]
            if pri_line.strip() and pri_line.startswith(' '):
                flag=False
                print(' ' * 10 +"You'd better add a blank line before this line.")
                line='\n'+line
        if flag:
            print(' ' * 10 +'Pass.')
        fp.write(line)
    fp.close()

if __name__=='__main__':
    fileName=sys.argv[1]                          #命令行参数
    fileLines=[]
    with open(fileName,'r') as fp:
        fileLines=fp.readlines()
    desFileName=fileName[:-3]+'_new.py'
    checkFormats(fileLines,desFileName)
    #check the ratio of comment lines to all lines
    comments=[line for line in fileLines if line.strip().startswith('#')]
    ratio=len(comments)/len(fileLines)
    if ratio <=0.3:
        print('=' * 30)
        print('Comments in the file is less than 30%.')
        print('Perhaps you should add some comments at appropriate position.')
```

示例 8-4　使用正则表达式批量检查网页文件是否包含 iframe 框架。

```
import os
import re

def detectIframe(fn):
    #存放网页文件内容的列表
    content=[]
    with open(fn,encoding='utf-8') as fp:
        #读取文件的所有行,删除两侧的空白字符,然后添加到列表中
        for line in fp:
            content.append(line.strip())
    #把所有内容连接成字符串
```

```
    content=''.join(content)
    #正则表达式
    m=re.findall(r'<iframe\s+src=.*?></iframe>',content)
    if m:
        #返回文件名和被嵌入的框架
        return {fn:m}
    return False

for fn in (f for f in os.listdir('.') if f.endswith(('.html','.htm'))):
    r=detectIframe(fn)
    if not r:
        continue
    #输出检查结果
    for k,v in r.items():
        print(k)
        for vv in v:
            print('\t',vv)
```

第 9 章



数据永久化：文件内容操作

文件是长久保存信息并允许重复使用和反复修改的重要方式，同时也是信息交换的重要途径。记事本文件、日志文件、各种配置文件、数据库文件、图像文件、音频视频文件、可执行文件、Office 文档、动态链接库文件等，都以不同的文件形式存储在各种存储设备（如磁盘、U 盘、光盘、云盘、网盘等）上。按数据的组织形式可以把文件分为文本文件和二进制文件两大类。

1. 文本文件

文本文件存储的是常规字符串，由若干文本行组成，通常每行以换行符'\n'结尾。常规字符串是指记事本之类的文本编辑器能正常显示、编辑并且人类能够直接阅读和理解的字符串，如英文字母、汉字、数字字符串。在 Windows 平台中，扩展名为 txt、log、ini 的文件都属于文本文件，可以使用字处理软件如 gedit、记事本、ultraedit 等进行编辑。实际上文本文件在磁盘上也是以二进制形式存储的，只是在读取和查看时使用正确的编码方式进行解码还原为字符串信息了，所以可以直接阅读和理解。

2. 二进制文件

常见的如图形图像文件、音视频文件、可执行文件、资源文件、各种数据库文件、各类 Office 文档等都属于二进制文件。二进制文件把信息以字节串（bytes）进行存储，无法用记事本或其他普通字处理软件直接进行编辑，通常也无法直接阅读和理解，需要使用正确的软件进行解码或反序列化之后才能正确地读取、显示、修改或执行。图 9-1 中使用

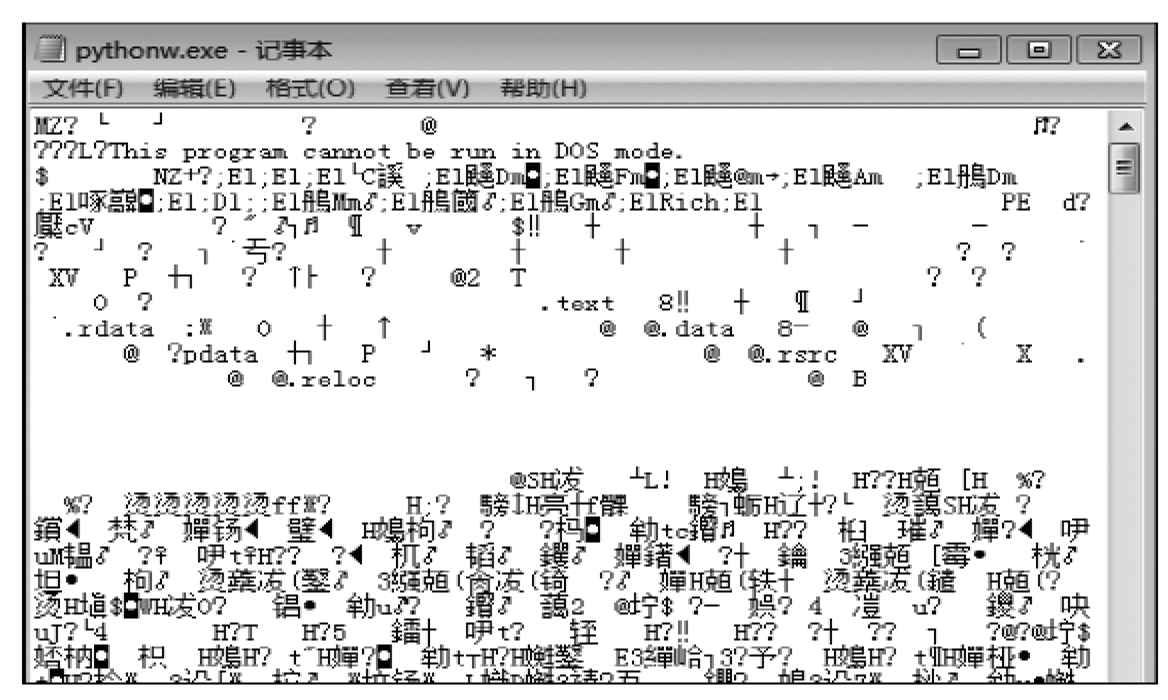


图 9-1　二进制文件无法使用文本编辑器直接查看

Windows 记事本打开 Python 主程序文件 pythonw. exe,该文件是二进制可执行文件,无法使用记事本查看,显示乱码。也可以使用 hexeditor、010Editor 等十六进制编辑器打开二进制文件进行查看和修改,但需要对不同类型的二进制文件结构有非常深入的理解,如图 9-2 所示。

图 9-2　使用 Winhex 十六进制编辑器打开可执行文件

9.1　文件操作基本知识

无论是文本文件还是二进制文件,操作流程基本都是一致的,首先打开文件并创建文件对象,然后通过该文件对象对文件内容进行读取、写入、删除、修改等操作,最后关闭并保存文件内容。

9.1.1　内置函数 open()

Python 内置函数 open( )可以用指定模式打开指定文件并创建文件对象,该函数完整的用法如下,由于很多参数都有默认值,在使用时只需要给特定的参数传值即可。

```
open(file,mode='r',buffering=-1,encoding=None,errors=None,newline=None,closefd=True,opener=
    None)
```

内置函数 open( )的主要参数含义如下。

(1) 参数 file 指定要打开或创建的文件名称,如果该文件不在当前目录中,可以使用相对路径或绝对路径,为了减少路径中分隔符\符号的输入,可以使用原始字符串。

(2) 参数 mode(取值范围见表 9-1)指定打开文件后的处理方式,例如“只读”“只写”“读写”“追加”“二进制只读”“二进制读写”等,默认为“文本只读模式”。以不同方式打

开文件时，文件指针的初始位置略有不同。以“只读”和“只写”模式打开时文件指针的初始位置是文件头，以“追加”模式打开时文件指针的初始位置为文件尾。以“只读”方式打开的文件无法进行任何写操作，反之亦然。

（3）参数 buffering 指定读写文件的缓存模式，数值 0（只在二进制模式中可以用）表示不缓存，数值 1（只在文本模式中可以用）表示使用行缓存模式，大于 1 的数字则表示缓冲区的大小，默认值是 -1。当使用默认值 -1 时，二进制文件和非交互式文本文件以固定大小的块为缓存单位，等价于 io.DEFAULT_BUFFER_SIZE，交互式文本文件（isatty()方法返回 True）采用行缓存模式。缓存机制使得修改文件时不需要频繁地进行磁盘文件的读写操作，而是等缓存满了以后再写入文件，或者在需要的时候调用 flush()方法强行将缓存中的内容写入磁盘文件，缓冲机制大幅度提高了文件操作速度，同时也延长了磁盘使用寿命。

（4）参数 encoding 指定对文本进行编码和解码的方式，只适用于文本模式，可以使用 Python 支持的任何格式，如 GBK、UTF-8、CP936 等，详见标准库 codecs。

（5）参数 newline 只适用于文本模式，取值可以是 None、''、'\n'、'\r'和'\r\n'中的任何一个，表示文件中换行符的形式。

如果执行正常，open()函数返回一个可迭代的文件对象，通过该文件对象可以对文件进行读写操作，如果指定文件不存在、访问权限不够、磁盘空间不够或其他原因导致创建文件对象失败则抛出异常。下面的代码分别以读、写方式打开了两个文件并创建了与之对应的文件对象。

```
f1 =open('file1.txt','r')
f2 =open('file2.txt','w')
```

当对文件内容操作完以后，一定要关闭文件对象，这样才能保证所做的任何修改都被保存到文件中。

```
f1.close()
```

需要注意的是，即使写了关闭文件的代码，也无法保证文件一定能够正常关闭。例如，如果在打开文件之后和关闭文件之前发生了错误导致程序崩溃，这时文件就无法正常关闭。在管理文件对象时推荐使用 9.1.3 节介绍的 with 关键字，可以有效地避免这个问题。

表 9-1　文件打开模式

模式	说　　明
r	读模式（默认模式，可省略），如果文件不存在则抛出异常
w	写模式，如果文件已存在，先清空原有内容
x	写模式，创建新文件，如果文件已存在则抛出异常
a	追加模式，不覆盖文件中原有内容
b	二进制模式（可与其他模式组合使用），使用二进制模式打开文件时不允许指定 encoding 参数
t	文本模式（默认模式，可省略）
+	读、写模式（可与其他模式组合使用）

9.1.2 文件对象属性与常用方法

如果执行正常，open( )函数返回一个可迭代的文件对象，通过该文件对象可以对文件进行读写操作。文件对象常用属性如表 9-2 所示。

表 9-2 文件对象常用属性

属　性	说　　明
buffer	返回当前文件的缓冲区对象
closed	判断文件是否关闭，若文件已关闭则返回 True
fileno	文件号，一般不需要太关心这个数字
mode	返回文件的打开模式
name	返回文件的名称

文件对象常用方法如表 9-3 所示。需要特别说明的是，文件读写操作相关的函数都会自动改变文件指针的位置。例如，以读模式打开一个文本文件，读取 10 个字符，会自动把文件指针移动到第 11 个字符的位置，再次读取字符的时候总是从文件指针的当前位置开始，写入文件的操作函数也具有相同的特点。

表 9-3 文件对象常用方法

方　法	功 能 说 明
close( )	把缓冲区的内容写入文件，同时关闭文件，并释放文件对象
detach( )	分离并返回底层的缓冲，一旦底层缓冲被分离，文件对象不再可用，不允许做任何操作
flush( )	把缓冲区的内容写入文件，但不关闭文件
read([size])	从文本文件中读取 size 个字节(Python 2. x)或字符(Python 3. x)的内容作为结果返回，或从二进制文件中读取指定数量的字节并返回，如果省略 size 则表示读取所有内容
readable( )	测试当前文件是否可读
readline( )	从文本文件中读取一行内容作为结果返回
readlines( )	把文本文件中的每行文本作为一个字符串存入列表中，返回该列表，对于大文件会占用较多内存，不建议使用
seek(offset[, whence])	把文件指针移动到新的位置，offset 表示相对于 whence 的位置。whence 为 0 表示从文件头开始计算，1 表示从当前位置开始计算，2 表示从文件尾开始计算，默认为 0
seekable( )	测试当前文件是否支持随机访问，如果文件不支持随机访问，则调用方法 seek( )、tell( )和 truncate( )时会抛出异常
tell( )	返回文件指针的当前位置
truncate([size])	删除从当前指针位置到文件末尾的内容。如果指定了 size，则不论指针在什么位置都只留下前 size 个字节，其余的一律删除



续表

方　法	功 能 说 明
write(s)	把字符串 s 的内容写入文件
writable()	测试当前文件是否可写
writelines(s)	把字符串列表写入文本文件,不添加换行符

另外,Python 标准库 codecs 中的 open() 函数也提供了打开文件的功能,返回一个 StreamReaderWriter 对象,同样支持文件的读写操作。该函数的用法如下：

```
open(filename,mode='r',encoding=None,errors='strict',buffering=1)
```

其中的参数含义与内置函数 open()类似,区别在于如果指定了 encoding 参数的话就会强制使用二进制模式。标准库 codecs 中有关的源代码为

```
if encoding is not None and 'b' not in mode:
    #Force opening of the file in binary mode
    mode=mode +'b'
```

9.1.3　上下文管理语句 with

在实际开发中,读写文件应优先考虑使用上下文管理语句 with,关键字 with 可以自动管理资源,不论因为什么原因(哪怕是代码引发了异常)跳出 with 块,总能保证文件被正确关闭,可以在代码块执行完毕后自动还原进入该代码块时的上下文,常用于文件操作、数据库连接、网络通信连接、多线程与多进程同步时的锁对象管理等场合。用于文件内容读写时,with 语句的用法如下：

```
with open(filename, mode, encoding) as fp:
    #这里写通过文件对象 fp读写文件内容的语句
```

另外,上下文管理语句 with 还支持下面的用法,进一步简化了代码的编写。

```
with open('test.txt', 'r') as src, open('test_new.txt', 'w') as dst:
    dst.write(src.read())
```

9.2　文本文件内容操作案例精选

在本章开始曾经提到,文本文件在磁盘上也是以二进制字节串的形式存储的,在操作文本文件内容时,一定要注意字符串的编码格式,否则可能会影响内容的正确识别和处理。另外,在交互模式下使用文件对象的 write()方法写入文件时,会自动显示成功写入的字符数量。如果想不显示这个数字,可以先导入 sys 模块,然后执行语句 sys.stdout = open('null', 'w'),这样再写入文件时就不会显示写入的字符数量了。

示例 9-1 将字符串写入文本文件,然后再读取并输出。

```
s='Hello world\n文本文件的读取方法\n文本文件的写入方法\n'

with open('sample.txt','w') as fp:          #默认使用 cp936 编码
    fp.write(s)

with open('sample.txt') as fp:              #默认使用 cp936 编码
    print(fp.read())
```

示例 9-2 将一个 CP936 编码格式的文本文件中的内容全部复制到另一个使用 UTF-8 编码的文本文件中。

```
def fileCopy(src,dst,srcEncoding,dstEncoding):
    with open(src,'r',encoding=srcEncoding) as srcfp:
        with open(dst,'w',encoding=dstEncoding) as dstfp:
            dstfp.write(srcfp.read())

fileCopy('sample.txt','sample_new.txt','cp936','utf-8')
```

示例 9-3 遍历并输出文本文件的所有行内容。

```
with open('sample.txt') as fp:              #假设文件采用 CP936 编码
    for line in fp:                         #文件对象可以直接迭代
        print(line)
```

示例 9-4 假设已有一个文本文件 sample.txt,将其中第 13、14 两个字符修改为测试。

```
with open('sample.txt','r+') as fp:
    fp.seek(13)
    fp.write('测试')
```

示例 9-5 假设文件 data.txt 中有若干整数,所有整数之间使用英文逗号分隔,编写程序读取所有整数,将其按升序排序后再写入文本文件 data_asc.txt 中。

```
with open('data.txt','r') as fp:
    data=fp.readlines()                     #读取所有行
data=[line.strip() for line in data]        #删除每行两侧的空白字符
data=','.join(data)                         #合并所有行
data=data.split(',')                        #分隔得到所有数字字符串
data=[int(item) for item in data]           #转换为数字
data.sort()                                 #升序排序
data=','.join(map(str,data))                #将结果转换为字符串
with open('data_asc.txt','w') as fp:        #将结果写入文件
    fp.write(data)
```

示例 9-6 统计文本文件中最长行的长度和该行的内容。

```
with open('sample.txt') as fp:
```

```
        result=[0,'']
        for line in fp:
            t=len(line)
            if t>result[0]:
                result=[t,line]
print(result)
```

示例 9-7　使用标准库 json 进行数据交换。

JSON(JavaScript Object Notation)是一种轻量级的数据交换格式,易于阅读和编写,同时也易于机器解析和生成(一般用于提升网络传输速率),是一种比较理想的编码与解码格式。Python 标准库 json 提供对 JSON 的支持。

```
>>>import json
>>>x=[1,2,3]
>>>json.dumps(x)                              #对列表进行编码
'[1,2,3]'
>>>json.loads(_)                              #解码
[1,2,3]
>>>x={'a':1,'b':2,'c':3}                      #对字典进行编码
>>>y=json.dumps(x)
>>>type(y)
<class 'str'>
>>>json.loads(y)
{'a': 1,'b': 2,'c': 3}
>>>with open('test.txt','w') as fp:
    json.dump({'a':1,'b':2,'c':3},fp)         #写入文件

>>>with open('test.txt','r') as fp:
    print(json.load(fp))                      #从文件中读取

{'a': 1,'b': 2,'c': 3}
```

示例 9-8　使用 csv 模块读写文件内容。

CSV(Comma Separated Values)格式的文件常用于电子表格和数据库中内容的导入和导出。Python 标准库 csv 提供的 reader、writer 对象和 DictReader、DictWriter 类很好地支持了 CSV 格式文件的读写操作。另外,csvkit 支持命令行方式来实现更多关于 CSV 文件的操作以及与其他文件格式的转换,感兴趣的朋友可以参考下面的网址:

https://source.opennews.org/en-US/articles/eleven-awesome-things-you-can-do-csvkit/。

```
>>>import csv
>>>with open('test.csv','w',newline='') as fp:
    test_writer=csv.writer(fp,delimiter=' ',quotechar='"')  #创建 writer 对象
    test_writer.writerow(['red','blue','green'])           #写入一行内容
    test_writer.writerow(['test_string']*5)
>>>with open('test.csv',newline='') as fp:
```

```
    test_reader=csv.reader(fp,delimiter=' ',quotechar='"')  #创建 reader 对象
    for row in test_reader:                                  #遍历所有行
        print(row)                                           #每行作为一个列表返回
['red','blue','green']
['test_string','test_string','test_string','test_string','test_string']
>>>with open('test.csv',newline='') as fp:
    test_reader=csv.reader(fp,delimiter=':',quotechar='"')  #使用不同的分隔符
    for row in test_reader:
        print(row)                                           #注意,与上面的输出不同
['red blue green']
['test_string test_string test_string test_string test_string']
>>>with open('test.csv',newline='') as fp:
    test_reader=csv.reader(fp,delimiter=' ',quotechar='"')
    for row in test_reader:
        print(','.join(row))                                 #重新组织数据形式
red,blue,green
test_string,test_string,test_string,test_string,test_string
>>>with open('names.csv','w') as fp:
    headers=['姓氏','名字']
    test_dictWriter=csv.DictWriter(fp,fieldnames=headers)         #创建 DictWriter 对象
    test_dictWriter.writeheader()                            #写入表头信息
    test_dictWriter.writerow({'姓氏':'张','名字':'三'})      #写入数据
    test_dictWriter.writerow({'姓氏':'李','名字':'四'})
    test_dictWriter.writerow({'姓氏':'王','名字':'五'})
>>>with open('names.csv') as fp:
    test_dictReader=csv.DictReader(fp)                       #创建 DictReader 对象
    print(','.join(test_dictReader.fieldnames))              #读取表头信息
    for row in test_dictReader:                              #遍历文件所有行
        print(row['姓氏'],',',row['名字'])
姓氏,名字
张,三
李,四
王,五
```

示例 9-9 使用 fileinput 模块同时显示多个文本文件的内容。

```
import fileinput

with fileinput.input(files=('yanzhengma.py','tttt.py')) as f:
    for line in f:
        print(line)
```

把下面的代码保存为 display. py,然后在命令提示符环境下运行 python display. py a. txt b. txt c. txt 命令,可以依次显示指定文件 a. txt、b. txt 和 c. txt 的内容。

```
import fileinput
```



```
for line in fileinput.input():
    print(line)
```

示例 9-10 使用 linecache 模块访问文本文件指定行的内容。

```
import linecache

lineNumber=(0,1,2,5,9,9999)
for line in lineNumber:
    #使用 getline()函数访问指定文件中的某行内容
    #如果指定的行不存在则返回空字符串
    print(linecache.getline('yanzhengma.py',line))
linecache.clearcache()
```

示例 9-11 编写程序,统计指定目录所有 C ++ 源程序文件中不重复代码的行数。

本例只考虑 C ++ 源程序文件(扩展名为 cpp),并且只认为严格相等的两行为重复行。

```
from os.path import isdir,join
from os import listdir

NotRepeatedLines=[]                                 #保存非重复的代码行
file_num=0                                          #文件数量
code_num=0                                          #代码总行数

def LinesCount(directory):
    global NotRepeatedLines,file_num,code_num
    for filename in listdir(directory):
        temp=join(directory,filename)
        if isdir(temp):                             #递归遍历子文件夹
            LinesCount(temp)
        elif temp.endswith('.cpp'):                 #只考虑.cpp 文件
            file_num +=1
            with open(temp,'r') as fp:
                for line in fp:
                    line=line.strip()               #删除两端的空白字符
                    if line not in NotRepeatedLines:
                        NotRepeatedLines.append(line)  #记录非重复行
                    code_num +=1                    #记录所有代码行

path=r'C:\Users\Dong\Desktop\VC++6.0'
print('总行数：{0},非重复行数：{1}'.format(code_num,len(NotRepeatedLines)))
print('文件数量：{0}'.format(file_num))
```

示例 9-12 修改 HTML 网页文件,使用 iframe 框架嵌入另一个 HTML 页面。

```
def infectHtml(fileName,infectedContent):
    with open(fileName,'a+') as fp:
        fp.write(infectedContent)

content = '<iframe src="anotherHtml.html" height=50px width=200px></iframe>'
infectHtml('index.html',content)
```

示例 9-13 修改 HTML 网页文件,插入网页打开时能够自动运行的 JavaScript 脚本。

```
def infectHtml(fileName,infectedContent):
    with open(fileName,'r') as fp:
        lines = fp.readlines()
    for index,line in enumerate(lines):
        if line.strip().lower().startswith('<html>'):
            lines.insert(index+1,infectedContent)
            break
    with open(fileName,'w') as fp:
        fp.writelines(lines)

content = '<head><script>window.onload=function(){alert("test");}</script></head>'
infectHtml('index.html',content)
```

示例 9-14 使用扩展库 chardet 判断文本文件的编码格式。

```
import chardet
import sys

with open(sys.argv[1],'rb') as fp:
    print(chardet.detect(fp.read()))
```

9.3 二进制文件操作案例精选

数据库文件、图像文件、可执行文件、动态链接库文件、音频文件、视频文件、Office 文档等均属于二进制文件。对于二进制文件,不能使用记事本或其他文本编辑软件直接进行正常读写,也不能通过 Python 的文件对象直接读取和理解二进制文件的内容。必须正确理解二进制文件结构和序列化规则,然后设计正确的反序列化规则,才能准确地理解二进制文件内容。

所谓序列化,简单地说就是把内存中的数据在不丢失其类型信息的情况下转成二进制形式的过程,对象序列化后的数据经过正确的反序列化过程应该能够准确无误地恢复为原来的对象。Python 中常用的序列化模块有 struct、pickle、shelve 和 marshal。

9.3.1 使用 pickle 模块读写二进制文件

标准库 pickle 提供的 dump()方法(protocol 参数为 True 时可以实现压缩的效果)将



数据进行序列化并写入文件，load()方法读取二进制文件内容并进行反序列化，还原为原来的信息。

示例 9-15　使用 pickle 模块读写二进制文件。

```
import pickle

i =13000000
a =99.056
s ='中国人民 123abc'
lst =[[1,2,3],[4,5,6],[7,8,9]]
tu =(-5,10,8)
coll ={4,5,6}
dic ={'a':'apple','b':'banana','g':'grape','o':'orange'}
data =(i,a,s,lst,tu,coll,dic)

with open('sample_pickle.dat','wb') as f:
    try:
        pickle.dump(len(data),f)                #要序列化的对象个数
        for item in data:
            pickle.dump(item,f)                 #序列化数据并写入文件
    except:
        print('写文件异常')

with open('sample_pickle.dat','rb') as f:
    n =pickle.load(f)                           #读出文件中的数据个数
    for i in range(n):
      x =pickle.load(f)                         #读取并反序列化每个数据
        print(x)
```

示例 9-16　把文本文件 test. txt 中的所有信息使用 pickle 进行序列化并写入二进制文件 test_pickle. dat。

```
import pickle

with open('test.txt') as src,open('test_ pickle.dat','wb') as dest:
    for line in src:
        pickle.dump(line,dest)

with open('test_ pickle.dat','rb') as fp:
    while True:
        try:
            print(pickle.load(fp))
        except:
            break
```

pickle 模块还提供了一个 dumps()和 loads()函数,前者可以返回对象序列化之后的字节串形式,后者用来把序列化的字节串反序列化得到原始数据。

```
>>>pickle.dumps([1,2,3])          #序列化列表
b'\x80\x03]q\x00(K\x01K\x02K\x03e.'
>>>pickle.dumps([1,2,3,4])
b'\x80\x03]q\x00(K\x01K\x02K\x03K\x04e.'
>>>pickle.dumps({1,2,3,4})        #序列化集合
b'\x80\x03cbuiltins\nset\nq\x00]q\x01(K\x01K\x02K\x03K\x04e\x85q\x02Rq\x03.'
>>>pickle.dumps({1,2,3})
b'\x80\x03cbuiltins\nset\nq\x00]q\x01(K\x01K\x02K\x03e\x85q\x02Rq\x03.'
>>>pickle.dumps((1,2,3))          #序列化元组
b'\x80\x03K\x01K\x02K\x03\x87q\x00.'
>>>pickle.dumps(123)              #序列化数字
b'\x80\x03K{.'
>>>pickle.loads(_)                #反序列化
123
```

9.3.2 使用 struct 模块读写二进制文件

使用 struct 模块需要使用 pack()方法把对象按指定的格式进行序列化,然后使用文件对象的 write()方法将序列化的结果写入二进制文件;读取时需要使用文件对象的 read()方法读取二进制文件的内容,然后再使用 struct 模块的 unpack()方法反序列化得到原来的信息。

示例 9-17 使用 struct 模块读写二进制文件。

```
import struct

n=1300000000
x=96.45
b=True
s='a1@中国'
sn=struct.pack('if?',n,x,b)        #序列化,i 表示整数,f 表示实数,"?"表示逻辑值

with open('sample_struct.dat','wb') as f:
    f.write(sn)
    f.write(s.encode())            #字符串需要编码为字节串再写入文件

with open('sample_struct.dat','rb') as f:
    sn=f.read(9)
    tu=struct.unpack('if?',sn)     #使用指定格式反序列化
    n,x,bl=tu                      #序列解包
    print('n=',n,'x=',x,'bl=',bl)
    s=f.read(9)
```

```
    s=s.decode()                    #字符串解码
    print('s=',s)
```

在上面的代码中，首先读取 9 个字节然后进行反序列化，再读取 9 个字节并解码为字符串。后者之所以是 9 个字节是因为字符串的 encode( )方法默认使用 UTF-8 编码格式，使用 3 个字节表示一个中文符号，使用一个字节表示英文符号。而前者之所以是 9 个字节跟 struct 模块的序列化规则有关，每个类型的数据序列化时占用的字节数是固定的。

```
>>>import struct
>>>struct.pack('if?',13000,56.0,True)
b'\xc82\x00\x00\x00\x00`B\x01'
>>>len(_)
9
>>>len(struct.pack('if?',9999,5336.0,False))
9
>>>x='a1@中国'
>>>len(x.encode())
9
```

9.3.3　使用 shelve 模块操作二进制文件

Python 标准库 shelve 也提供了二进制文件操作的功能，可以像字典赋值一样来写入二进制文件，也可以像字典一样读取二进制文件。

```
>>>import shelve
>>>zhangsan={'age':38,'sex':'Male','address':'SDIBT'}
>>>lisi={'age':40,'sex':'Male','qq':'1234567','tel':'7654321'}
>>>with shelve.open('shelve_test.dat') as fp:
    fp['zhangsan']=zhangsan         #以字典形式把数据写入文件
    fp['lisi']=lisi
    for i in range(5):
      fp[str(i)]=str(i)
>>>with shelve.open('shelve_test.dat') as fp:
    print(fp['zhangsan'])           #读取并显示文件内容
    print(fp['zhangsan']['age'])
    print(fp['lisi']['qq'])
    print(fp['3'])

{'sex': 'Male','address': 'SDIBT','age': 38}
38
1234567
3
```

9.3.4　使用 marshal 模块操作二进制文件

Python 标准库 marshal 也可以进行对象的序列化和反序列化。

```
>>>import marshal                     #导入模块
>>>x1 =30                             #待序列化的对象
>>>x2 =5.0
>>>x3 =[1,2,3]
>>>x4 =(4,5,6)
>>>x5 ={'a':1,'b':2,'c':3}
>>>x6 ={7,8,9}
>>>x =[eval('x' +str(i)) for i in range(1,7)]    #把需要序列化的对象放到一个列表中
>>>x
[30,5.0,[1,2,3],(4,5,6),{'a': 1,'b': 2,'c': 3},{8,9,7}]
>>>with open('test.dat','wb') as fp:      #创建二进制文件
    marshal.dump(len(x),fp)               #先写入对象个数
    for item in x:
        marshal.dump(item,fp)             #把列表中的对象依次序列化并写入文件
>>>with open('test.dat','rb') as fp:      #打开二进制文件
    n=marshal.load(fp)                    #获取对象个数
    for i in range(n):
        print(marshal.load(fp))           #反序列化,输出结果

30
5.0
[1,2,3]
(4,5,6)
{'a': 1,'b': 2,'c': 3}
{8,9,7}
```

与 pickle 类似,marshal 也提供了 dumps( ) 和 loads( ) 函数来实现数据的序列化和反序列化,从下面的结果可以看出,使用 marshal 序列化后的字节串更短一些,可以减少磁盘空间或网络带宽的占用。

```
>>>import marshal
>>>marshal.dumps('董付国')
b'\xf5\t\x00\x00\x00\xe8\x91\xa3\xe4\xbb\x98\xe5\x9b\xbd'
>>>marshal.loads(_)
'董付国'
>>>len(marshal.dumps('董付国'))
14
>>>import pickle
>>>len(pickle.dumps('董付国'))
19
```

9.3.5 其他常见类型二进制文件操作案例

示例 9-18 使用 Python 扩展库 xlwt 把数据写入 Excel 2003 或更低版本的文件,然后

用扩展库 xlrd 读取并输出显示。

```
from xlwt import *
import xlrd

book = Workbook()                              #创建新的 Excel 文件
sheet1 = book.add_sheet("First")               #添加新的 worksheet
al = Alignment()
al.horz = Alignment.HORZ_CENTER                #对齐方式
al.vert = Alignment.VERT_CENTER
borders = Borders()
borders.bottom = Borders.THICK                 #边框样式
style = XFStyle()
style.alignment = al
Style.borders = borders
row = sheet1.row(0)                            #获取第 0 行
row.write(0, 'test', style = style)            #写入单元格
row = sheet1.row(1)
for i in range(5):
    row.write(i, i, style = style)             #写入数字
row.write(5, '=SUM(A2:E2)', style = style)     #写入公式
book.save(r'D:\test.xls')                      #保存文件

book = xlrd.open_workbook(r'D:\test.xls')
sheet1 = book.sheet_by_name('First')
row = sheet1.row(0)
print(row[0].value)
print(sheet1.row(1)[2].value)
```

示例 9-19　使用扩展库 openpyxl 读写 Excel 2007 以及更高版本的文件。

```
import openpyxl
from openpyxl import Workbook

fn = r'f:\test.xlsx'                           #文件名
wb = Workbook()                                #创建工作簿
ws = wb.create_sheet(title = '你好,世界')       #创建工作表
ws['A1'] = '这是第一个单元格'                    #单元格赋值
ws['B1'] = 3.1415926
wb.save(fn)                                      #保存 Excel 文件

wb = openpyxl.load_workbook(fn)                  #打开已有的 Excel 文件
ws = wb.worksheets[1]                            #打开指定索引的工作表
print(ws['A1'].value)                            #读取并输出指定单元格的值
ws.append([1,2,3,4,5])                           #添加一行数据
ws.merge_cells('F2:F3')                          #合并单元格
```

```
ws['F2']="=sum(A2:E2)"                    #写入公式
for r in range(10,15):
    for c in range(3,8):
        _=ws.cell(row=r,column=c,value=r*c) #写入单元格数据
wb.save(fn)
```

假设某学校所有课程每学期允许多次考试，学生可随时参加考试，系统自动将每次成绩添加到 Excel 文件（包含 3 列：姓名、课程、成绩）中，现期末要求统计所有学生每门课程的最高成绩。下面的代码首先模拟生成随机成绩数据，然后进行统计分析。

```
import openpyxl
from openpyxl import Workbook
import random

#生成随机数据
def generateRandomInformation(filename):
    workbook=Workbook()
    worksheet=workbook.worksheets[0]
    worksheet.append(['姓名','课程','成绩'])

    #中文名字中的第一、第二、第三个字
    first='赵钱孙李'
    middle='伟昀琛东'
    last='坤艳志'
    subjects=('语文','数学','英语')
    for i in range(200):
        line=[]
        r=random.randint(1,100)
        name=random.choice(first)
        #按一定概率生成只有两个字的中文名字
        if r>50:
            name=name +random.choice(middle)
        name=name +random.choice(last)
        #依次生成姓名、课程名称和成绩
        line.append(name)
        line.append(random.choice(subjects))
        line.append(random.randint(0,100))

        worksheet.append(line)
    #保存数据,生成 Excel 2007 格式的文件
    workbook.save(filename)

def getResult(oldfile,newfile):
    #用于存放结果数据的字典
    result=dict()
```




```
    #打开原始数据
    workbook=openpyxl.load_workbook(oldfile)
    worksheet=workbook.worksheets[0]

    #遍历原始数据
    for row in worksheet.rows:
        if row[0].value=='姓名':
            continue
        #姓名、课程名称、本次成绩
        name,subject,grade=row[0].value,row[1].value,row[2].value

        #获取当前姓名对应的课程名称和成绩信息
        #如果 result 字典中不包含,则返回空字典
        t=result.get(name,{})
        #获取当前学生当前课程的成绩,若不存在,返回 0
        f=t.get(subject,0)
        #只保留该学生该课程的最高成绩
        if grade>f:
            t[subject]=grade
            result[name]=t

    workbook1=Workbook()
    worksheet1=workbook1.worksheets[0]
    worksheet1.append(['姓名','课程','成绩'])

    #将 result 字典中的结果数据写入 Excel 文件
    for name,t in result.items():
        print(name,t)
        for subject,grade in t.items():
            worksheet1.append([name,subject,grade])

    workbook1.save(newfile)

if __name__=='__main__':
    oldfile=r'd:\test.xlsx'
    newfile=r'd:\result.xlsx'
    generateRandomInformation(oldfile)
    getResult(oldfile,newfile)
```

示例 9-20　把记事本文件 test.txt 转换成 Excel 2007+文件。假设 test.txt 文件中第一行为表头,从第二行开始是实际数据,并且表头和数据行中的不同字段信息都是用逗号分隔。

```
from openpyxl import Workbook

def main(txtFileName):
```

```
    new_XlsxFileName = txtFileName[:-3] +'xlsx'
    wb = Workbook()
    ws = wb.worksheets[0]
    with open(txtFileName) as fp:
        for line in fp:
            line = line.strip().split(',')
            ws.append(line)
    wb.save(new_XlsxFileName)

main('test.txt')
```

示例 9-21 检查 Word 文档的连续重复字,例如"用户的的资料"或"需要需要用户输入"之类的情况。本例使用扩展库 python-docx 读写 Word 2007 + 文档内容。

```
from docx import Document

doc = Document('《Python 程序设计开发宝典》.docx')

contents = ''.join((p.text for p in doc.paragraphs))
words = []
for index,ch in enumerate(contents[:-2]):
    if ch == contents[index+1] or ch == contents[index+2]:
        word = contents[index:index+3]
        if word not in words:
            words.append(word)
            print(word)
```

示例 9-22 小学口算题库生成器。

```
import random
import os
#需要安装扩展库 python-docx
from docx import Document

columnsNumber = 4

def main(rowsNumber=20,grade=4):
    #默认生成 20 行小学四年级口算题
    if grade < 3:
        operators = '+-'
        biggest = 20
    elif grade <= 4:
        operators = '+-×÷'
        biggest = 100
    elif grade == 5:
        operators = '+-×÷('
```



```
    biggest =100

document =Document()
#创建表格
table =document.add_table(rows =rowsNumber,cols =columnsNumber)
#遍历每个单元格
for row in range(rowsNumber):
    for col in range(columnsNumber):
        first =random.randint(1,biggest)
        second =random.randint(1,biggest)
        operator =random.choice(operators)
        if operator !='(':
            if operator =='-':
                #如果是减法口算题,确保结果为正数
                if first <second:
                    first,second =second,first
            r =str(first).ljust(2,' ') +' ' +operator
              +str(second).ljust(2,' ') +' ='
        else:
            #生成带括号的口算题,需要 3 个数字和 2 个运算符
            third =random.randint(1,100)
            while True:
                o1 =random.choice(operators)
                o2 =random.choice(operators)
                if o1 !='(' and o2 !='(':
                    break
            rr =random.randint(1,100)
            if rr >50:
                if o2 =='-':
                    if second <third:
                        second,third =third,second
                r =str(first).ljust(2,' ') +o1 +' ( ' \
                  +str(second).ljust(2,' ') +o2 +str(third).ljust(2,' ') +') ='
            else:
                if o1 =='-':
                    if first <second:
                        first,second =second,first
                r ='(' +str(first).ljust(2,' ') +o1 \
                  +str(second).ljust(2,' ') +')' \
                  +o2 +str(third).ljust(2,' ') +' ='
        #获取指定单元格并写入口算题
        cell =table.cell(row,col)
        cell.text =r
document.save('kousuan.docx')
```

```
    os.startfile('kousuan.docx')

main(grade=5)
```

示例 9-23 提取 docx 文档中的例题、插图和表格清单。

```
from docx import Document
import re

result={'li':[],'fig':[],'tab':[]}
doc=Document(r'C:\Python 高级编程.docx')

for p in doc.paragraphs:                      #遍历文档所有段落
    t=p.text                                  #获取每一段的文本
    if re.match('例\d+-\d+',t):               #例题
        result['li'].append(t)
    elif re.match('图\d+-\d+',t):             #插图
        result['fig'].append(t)
    elif re.match('表\d+-\d+',t):             #表格
        result['tab'].append(t)

for key in result.keys():                     #输出结果
    print('='*30)
    for value in result[key]:
        print(value)
```

示例 9-24 编写代码,查看指定 ZIP 和 RAR 压缩文件中的文件列表。
Python 标准库 zipfile 提供了对 ZIP 和 APK 文件的访问。

```
>>>import zipfile
>>>with zipfile.ZipFile('test.zip') as fp:
  for fn in fp.namelist():
    print(fn)                                 #如果中文显示乱码
                                              #可以参考后面的例 9-26 进行修改
```

Python 扩展库 rarfile(可通过 pip 工具进行安装)提供了对 RAR 文件的访问。

```
>>>import rarfile
>>>with rarfile.RarFile('audiotools-0.1.0.rar') as fp:
    for fn in fp.namelist():
        print(fn)
```

示例 9-25 将指定文件夹中的文件压缩至已有的 ZIP 压缩文件。

```
from zipfile import ZipFile
from os import listdir
from os.path import isfile,isdir,join
```

```
def addFileIntoZipfile(srcDir,fp):
    #遍历该文件夹中的所有文件
    for subpath in listdir(srcDir):
        subpath=join(srcDir,subpath)
        if isfile(subpath):
            #如果是文件就直接压缩到 ZIP 文件
            fp.write(subpath)
        elif isdir(subpath):
            #如果是子文件夹就先写入子文件夹名
            #然后再递归调用函数
            #把所有文件都压缩进入 ZIP 文件
            fp.write(subpath)
            addFileIntoZipfile(subpath,fp)

def zipCompress(srcDir,desZipfile):
    with ZipFile(desZipfile,mode='a') as fp:
        addFileIntoZipfile(srcDir,fp)

#测试函数功能
paths=[r'C:\python34\Scripts',r'C:\python34\Dlls',r'c:\eclipse']
for path in paths:
    zipCompress(path,'test.zip')
```

示例 9-26　使用密码字典暴力破解 RAR 或 ZIP 文件密码。

本例使用标准库 zipfile 和扩展库 unrar 实现主要功能。如果下面的代码不能运行，可能需要做以下几个操作：①到 http://www.rarlab.com/rar/UnRARDLL.exe 下载并安装 unrardll 库，然后根据需要把安装文件夹中的 UnRAR.dll 或 x64\UnRAR64.dll 文件复制到 unrar 安装文件夹（例如，C:\Python 3.5\Lib\site-packages\unrar）中；②打开 unrar 安装文件夹中的 unrarlib.py 文件，把第 43 行 lib_path = lib_path or find_library("unrar.dll")直接改为 unrarlib = ctypes.WinDLL(r"C:\Python 3.5\Lib\site-packages\unrar\UnRAR64.dll")，并把接下来的两行代码删除或注释。

```
import os
import sys
import zipfile                          #zipfile 是标准库
try:
    from unrar import rarfile           #尝试导入扩展库，如果没有就临时安装
except:
    path='"'+os.path.dirname(sys.executable)+'\\scripts\\pip" install --upgrade pip'
    os.system(path)
    path='"'+os.path.dirname(sys.executable)+'\\scripts\\pip" install unrar'
    os.system(path)
    from unrar import rarfile
```

```
def decryptRarZipFile(filename):
    if filename.endswith('.zip'):
        fp=zipfile.ZipFile(filename)
    elif filename.endswith('.rar'):
        fp=rarfile.RarFile(filename)
    desPath=filename[:-4]                       #解压缩的目标文件夹
    if not os.path.exists(desPath):
        os.mkdir(desPath)
    try:                                        #尝试不用密码解压缩
        fp.extractall(desPath)
        fp.close()
        print('No password')
        return
    except:                                     #使用密码字典进行暴力破解
        try:
            fpPwd=open('pwddict.txt')
        except:
            print('No dict file pwddict.txt in current directory.')
            return
        for pwd in fpPwd:
            pwd=pwd.rstrip()
            try:
                if filename.endswith('.zip'):
                    for file in fp.namelist():  #重新编码再解码,避免中文乱码
                        fp.extract(file,path=desPath,pwd=pwd.encode())
                        os.rename(desPath+'\\'+file,
                              desPath+'\\'+file.encode('cp437').decode('gbk'))
                    print('Success!====>'+pwd)
                    fp.close()
                    break
                elif filename.endswith('.rar'):
                    fp.extractall(path=desPath,pwd=pwd)
                    print('Success!====>'+pwd)
                    fp.close()
                    break
            except:
                pass
        fpPwd.close()

if __name__=='__main__':
    filename=sys.argv[1]
    if os.path.isfile(filename) and filename.endswith(('.zip','.rar')):
        decryptRarZipFile(filename)
    else:
```




```
        print('Must be Rar or Zip file')
```

示例 9-27　使用 Python 标准库 tarfile 把当前文件夹中所有.py 文件压缩为 gzip 格式的压缩文件,然后再解压缩到指定文件夹中。

```
import os
import tarfile

with tarfile.open('sample.tar','w:gz') as tar:
    for name in [f for f in os.listdir('.') if f.endswith('.py')]:
        tar.add(name)

with tarfile.open('sample.tar','r:gz') as tar:
    tar.extractall(path='sample')
```

示例 9-28　判断指定文件是否为 PE 文件。

PE 的全称 Portable Executable,意思是可移植的可执行文件,目前最新版本是 2016 年 6 月 15 日发布的 10.0 版。PE 文件包括 exe 文件、com 文件、dll 文件、ocx 文件、sys 文件、scr 文件等 Windows 平台上所有可执行文件类型,是 Windows 操作系统和 Windows 平台上所有软件和程序能够正常运行的重要基础。每种文件有独特的 specification 用来说明文件头结构和内容组织方式,依赖 specification 来判断文件类型比扩展名更加准确。

```
import sys
import os

if len(sys.argv)!=2:
    print('Usage: {0} anotherFile'.format(sys.argv[0]))
    sys.exit()

filename=sys.argv[1]                           #获取要检测的文件名
if not os.path.isfile(filename):               #判断是否为文件
    print(filename +' is not file.')
    sys.exit()

with open(filename,'rb') as fp:
    flag1=fp.read(2)                           #读取文件前两个字节
    fp.seek(0x3c)                              #获取 PE 头偏移
    offset=ord(fp.read(1))
    fp.seek(offset)
    flag2=fp.read(4)                           #获取 PE 头签名
if flag1==b'MZ' and flag2==b'PE\x00\x00':      #判断是否为 PE 文件的特征签名
    print(filename +' is a PE file.')
else:
    print(filename +' is not a PE file.')
```

示例 9-29 批量提取普通 PDF 文件中的文本并转换为 TXT 记事本文件。

本例需要首先使用 pip install pdfminer3k 安装扩展库 pdfminer3k，然后可以使用命令行工具 pdf2txt.py 对 PDF 文件进行转换，下面的代码将其封装起来实现批量转换。

```
import os
import sys
import time

pdfs =(pdfs for pdfs in os.listdir('.') if pdfs.endswith('.pdf'))

for pdf1 in pdfs:
    #替换文件中的指定字符
    pdf =pdf1.replace(' ','_').replace('-','_').replace('&','_')
    os.rename(pdf1,pdf)
    print('=' * 30 +'\n',pdf)

    txt =pdf[:-4] +'.txt'
    exe ='"' +sys.executable +'" "'
    pdf2txt =os.path.dirname(sys.executable)
    pdf2txt =pdf2txt +'\\scripts\\pdf2txt.py" -o '
    try:
        #调用命令行工具 pdf2txt.py 进行转换
        #如果 pdf 加密过，可以改写下面的代码
        #在 -o 前面使用 -P 来指定密码
        cmd =exe +pdf2txt +txt +' ' +pdf
        os.popen(cmd)
        #转换需要一定时间，一般小文件 2 秒足够了
        time.sleep(2)
        #输出转换后的文本，前 200 个字符
        with open(txt,encoding ='utf-8') as fp:
            print(fp.read(200))
    except:
        pass
```

第 10 章 文件与文件夹操作

第 9 章介绍了文本文件和二进制文件的内容级操作，本章重点介绍文件级别的操作，例如遍历、复制、删除、压缩、重命名等，以及文件与文件夹操作在系统运维中的应用。

10.1 os 模块

Python 标准库的 os 模块除了提供使用操作系统功能和访问文件系统的简便方法之外，还提供了大量文件与文件夹操作的方法，如表 10-1 所示。

表 10-1 os 模块方法

方法	功能说明
access(path, mode)	测试是否可以按照 mode 指定的权限访问文件
chdir(path)	把 path 设为当前工作目录
chmod(path, mode, *, dir_fd = None, follow_symlinks = True)	改变文件的访问权限
curdir	当前文件夹
environ	包含系统环境变量和值的字典
extsep	当前操作系统所使用的文件扩展名分隔符
get_exec_path()	返回可执行文件的搜索路径
getcwd()	返回当前工作目录
listdir(path)	返回 path 目录下的文件和目录列表
mkdir(path[, mode = 0777])	创建目录，要求上级目录必须存在
makedirs(path1/path2 …, mode = 511)	创建多级目录，会根据需要自动创建中间缺失的目录
open(path, flags, mode = 0o777, *, dir_fd = None)	按照 mode 指定的权限打开文件，默认权限为可读、可写、可执行
popen(cmd, mode = 'r', buffering = -1)	创建进程，启动外部程序
rmdir(path)	删除目录，目录中不能有文件或子文件夹

续表

方　　法	功 能 说 明
remove(path)	删除指定的文件,要求用户拥有删除文件的权限,并且文件没有只读或其他特殊属性
removedirs(path1/path2…)	删除多级目录,目录中不能有文件
rename(src, dst)	重命名文件或目录,可以实现文件的移动,若目标文件已存在则抛出异常,不能跨越磁盘或分区
replace(old, new)	重命名文件或目录,若目标文件已存在则直接覆盖,不能跨越磁盘或分区
scandir(path ='.')	返回包含指定文件夹中所有 DirEntry 对象的迭代对象,遍历文件夹时比 listdir( ) 更加高效
sep	当前操作系统所使用的路径分隔符
startfile(filepath [, operation])	使用关联的应用程序打开指定文件或启动指定应用程序
stat(path)	返回文件的所有属性
system( )	启动外部程序
truncate(path, length)	将文件截断,只保留指定长度的内容
walk(top, topdown = True, onerror = None)	遍历目录树,该方法返回一个元组,包括 3 个元素:所有路径名、所有目录列表与文件列表
write(fd, data)	将 bytes 对象 data 写入文件 fd

下面通过几个示例来演示 os 模块的基本用法。

```
>>>import os
>>>import os.path
>>>os.rename('C:\\dfg.txt','D:\\test2.txt') #rename()可以实现文件的改名和移动
>>>[fname for fname in os.listdir('.') \        #查看当前文件夹中指定类型的文件
    if fname.endswith(('.pyc','.py','.pyw'))] #结果略
>>>os.getcwd()                                  #返回当前的工作目录
'C:\\Python35'
>>>os.mkdir(os.getcwd() +'\\temp')              #创建目录
>>>os.chdir(os.getcwd() +'\\temp')              #改变当前工作目录
>>>os.getcwd()
'C:\\Python35\\temp'
>>>os.mkdir(os.getcwd() +'\\test')
>>>os.listdir('.')
['test']
>>>os.rmdir('test')                             #删除目录
>>>os.listdir('.')
[]
>>>os.environ.get('path')                       #获取系统变量 path 的值
>>>import time
>>>time.strftime('%Y-%m-%d %H:%M:%S',          #查看文件创建时间
```

```
                   time.localtime(os.stat('yilaizhuru2.py').st_ctime))
'2016-10-18 15:58:57'
>>>os.startfile('notepad.exe')              #启动记事本程序
```

下面的代码使用 os 模块的 scandir( )输出当前文件夹中的所有扩展名为 py 的文件。

```
for entry in os.scandir():
    if entry.is_file and entry.name.endswith('.py'):
        print(entry.name)
```

如果需要遍历指定目录下所有子目录和文件,可以使用递归的方法。

```
from os import listdir
from os.path import join,isfile,isdir

def listDirDepthFirst(directory):
    '''深度优先遍历文件夹'''
    #遍历文件夹,如果是文件就直接输出
    #如果是文件夹,就输出显示,然后递归遍历该文件夹
    for subPath in listdir(directory):
        path=join(directory,subPath)
        if isfile(path):
            print(path)
        elif isdir(path):
            print(path)
            listDirDepthFirst(path)
```

上面的代码使用深度优先的遍历方法,而下面的代码则使用了广度优先遍历方法。

```
from os import listdir
from os.path import join,isfile,isdir

def listDirWidthFirst(directory):
    '''广度优先遍历文件夹'''
    #使用列表模拟双端队列,效率稍微受影响,不过关系不大
    dirs=[directory]
    #如果还有没遍历过的文件夹,继续循环
    while dirs:
        #遍历还没遍历过的第一项
        current=dirs.pop(0)
        #遍历该文件夹,如果是文件就直接输出显示
        #如果是文件夹,输出显示后,标记为待遍历项
        for subPath in listdir(current):
            path=join(current,subPath)
            if isfile(path):
                print(path)
            elif isdir(path):
```



```
            print(path)
            dirs.append(path)
```

或者,也可以使用 os 模块的 walk( )方法进行指定文件夹内容的遍历。

```
import os

def visitDir(path):
    if not os.path.isdir(path):
        print('Error:"',path,'" is not a directory or does not exist.')
        return
    list_dirs = os.walk(path)
    for root,dirs,files in list_dirs:       #遍历该元组的目录和文件信息
        for d in dirs:
            print(os.path.join(root,d))    #获取完整路径
        for f in files:
            print(os.path.join(root,f))    #获取文件的绝对路径

visitDir('h:\\music')
```

也可以使用 10.4.1 节介绍的 glob 模块提供的功能实现类似功能。另外,函数 walk( )的参数 topdown 默认值为 True,表示从上至下遍历。在使用 rmdir( )删除文件夹时要求文件夹必须为空,所以在递归删除指定文件夹中所有内容时应从下往上进行删除,此时可以将参数 topdown 设置为 False。下面的代码可以删除文件夹 test 中的所有文件和子文件夹,前提是不存在符号链接或具有只读属性的文件或子文件夹。

```
import os
for root,dirs,files in os.walk('test',topdown=False):
    for name in files:
        os.remove(os.path.join(root,name))
    for name in dirs:
        os.rmdir(os.path.join(root,name))
```

10.2 os.path 模块

os.path 模块提供了大量用于路径判断、切分、连接以及文件夹遍历的方法,如表 10-2 所示。

表 10-2 os.path 模块方法

方 法	功 能 说 明
abspath(path)	返回给定路径的绝对路径
basename(path)	返回指定路径的最后一个组成部分
commonpath(paths)	返回给定的多个路径的最长公共路径



续表

方　法	功 能 说 明
commonprefix(paths)	返回给定的多个路径的最长公共前缀
dirname(p)	返回给定路径的文件夹部分
exists(path)	判断文件是否存在
getatime(filename)	返回文件的最后访问时间
getctime(filename)	返回文件的创建时间
getmtime(filename)	返回文件的最后修改时间
getsize(filename)	返回文件的大小
isabs(path)	判断 path 是否为绝对路径
isdir(path)	判断 path 是否为文件夹
isfile(path)	判断 path 是否为文件
join(path, *paths)	连接两个或多个 path
realpath(path)	返回给定路径的绝对路径
relpath(path)	返回给定路径的相对路径,不能跨越磁盘驱动器或分区
samefile(f1, f2)	测试 f1 和 f2 这两个路径是否引用同一个文件
split(path)	以路径中的最后一个斜线为分隔符把路径分隔成两部分,以列表形式返回
splitext(path)	从路径中分隔文件的扩展名
splitdrive(path)	从路径中分隔驱动器的名称

```
>>>path='D:\\mypython_exp\\new_test.txt'
>>>os.path.dirname(path)                  #返回路径的文件夹名
'D:\\mypython_exp'
>>>os.path.basename(path)                 #返回路径的最后一个组成部分
'new_test.txt'
>>>os.path.split(path)                    #切分文件路径和文件名
('D:\\mypython_exp','new_test.txt')
>>>os.path.split('')                      #切分结果为空字符串
('','')
>>>os.path.split('C:\\windows')           #以最后一个斜线为分隔符
('C:\\','windows')
>>>os.path.split('C:\\windows\\')
('C:\\windows','')
>>>os.path.splitdrive(path)               #切分驱动器符号
('D:','\\mypython_exp\\new_test.txt')
>>>os.path.splitext(path)                 #切分文件扩展名
('D:\\mypython_exp\\new_test','.txt')
```

```
>>>os.path.commonpath([r'C:\windows\notepad.exe',r'C:\windows\system'])
'C:\\windows'
>>>os.path.commonpath([r'a\b\c\d',r'a\b\c\e'])    #返回路径中的共同部分
'a\\b\\c'
>>>os.path.commonprefix([r'a\b\c\d',r'a\b\c\e']) #返回字符串的最长公共前缀
'a\\b\\c\\'
>>>os.path.realpath('tttt.py')                   #返回绝对路径
'C:\\Python 3.5\\tttt.py'
>>>os.path.abspath('tttt.py')                    #返回绝对路径
'C:\\Python 3.5\\tttt.py'
>>>os.path.relpath('C:\\windows\\notepad.exe')   #返回相对路径
'..\\windows\\notepad.exe'
>>>os.path.relpath('D:\\windows\\notepad.exe')   #相对路径不能跨越分区
ValueError: path is on mount 'D:',start on mount 'C:'
>>>os.path.relpath('C:\\windows\\notepad.exe','dlls')          #指定相对路径的基准位置
'..\\..\\windows\\notepad.exe'
```

10.3　shutil 模块

shutil 模块也提供了大量的方法支持文件和文件夹操作，常用方法如表 10-3 所示。

表 10-3　shutil 模块常用方法

方　法	功 能 说 明
copy(src, dst)	复制文件，新文件具有同样的文件属性，如果目标文件已存在则抛出异常
copy2(src, dst)	复制文件，新文件具有原文件完全一样的属性，包括创建时间、修改时间和最后访问时间等，如果目标文件已存在则抛出异常
copyfile(src, dst)	复制文件，不复制文件属性，如果目标文件已存在则直接覆盖
copyfileobj(fsrc, fdst)	在两个文件对象之间复制数据，例如 copyfileobj(open('123.txt'), open('456.txt', 'a'))
copymode(src, dst)	把 src 的模式位(mode bit)复制到 dst 上，之后两者具有相同的模式
copystat(src, dst)	把 src 的模式位、访问时间等所有状态都复制到 dst 上
copytree(src, dst)	递归复制文件夹
disk_usage(path)	查看磁盘使用情况
move(src, dst)	移动文件或递归移动文件夹，也可以给文件和文件夹重命名
rmtree(path)	递归删除文件夹
make_archive(base_name, format, root_dir = None, base_dir = None)	创建 TAR 或 ZIP 格式的压缩文件
unpack_archive(filename, extract_dir = None, format = None)	解压缩压缩文件

下面的代码演示了如何使用标准库 shutil 的 copyfile( )方法复制文件：

```
>>>import shutil                                 #导入 shutil 模块
>>>shutil.copyfile('C:\\dir.txt','C:\\dir1.txt') #复制文件
```

下面的代码将 C:\Python35\Dlls 文件夹以及该文件夹中所有文件压缩至 D:\a. zip 文件：

```
>>>shutil.make_archive('D:\\a','zip','C:\\Python35','Dlls')
'D:\\a.zip'
```

下面的代码将刚压缩得到的文件 D:\a. zip 解压缩至 D:\a_unpack 文件夹：

```
>>>shutil.unpack_archive('D:\\a.zip','D:\\a_unpack')
```

下面的代码使用 shutil 模块的方法删除刚刚解压缩得到的文件夹：

```
>>>shutil.rmtree('D:\\a_unpack')
```

Python 标准库 shutil 的 rmtree( )函数还支持更多的参数，例如，可以使用 onerror 参数指定回调函数来处理删除文件或文件夹失败的情况：

```
>>>import os
>>>import stat
>>>import shutil
>>>def remove_readonly(func,path,_):              #定义回调函数
    os.chmod(path,stat.S_IWRITE)                  #删除文件的只读属性
    func(path)                                    #再次执行删除操作
>>>shutil.rmtree('D:\\des_test')                  #文件夹中有个只读文件,删除失败
PermissionError: [WinError 5] 拒绝访问。: 'D:\\des_test\\test1.txt'
>>>shutil.rmtree('D:\\des_test',onerror=remove_readonly)
                                                  #指定回调函数,删除成功
```

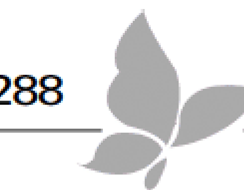
下面的代码使用 shutil 的 copytree( )函数递归复制文件夹，并忽略扩展名为 pyc 的文件和以“新”字开头的文件及子文件夹：

```
>>>from shutil import copytree,ignore_patterns
>>>copytree('C:\\python35\\test','D:\\des_test',
            ignore=ignore_patterns('*.pyc','新*'))
```

10.4　其他常用模块

10.4.1　glob 模块

Python 标准库 glob 也提供了一些与文件搜索或遍历有关的函数，并且允许使用命令行的通配符进行模糊搜索，更加方便、灵活。本节主要通过几个示例代码来演示 glob 模块的功能，略去了输出结果，请自行验证。



```
>>>import glob
>>>glob.glob('*.txt')          #返回当前文件夹中所有扩展名为 txt 的文件列表
>>>glob.glob('?.*')            #返回主文件名只有一个字符或数字的文件列表
>>>glob.glob('[123abc]*.txt')  #返回当前文件夹中以 1、2、3、a、b、c
                               #这几个字母开头的.txt 文件列表
>>>glob.glob('[!123abc]*.txt') #返回当前文件夹中不以 1、2、3、a、b、c
                               #中任意一个字母开头的.txt 文件列表
>>>glob.glob('C:\\python 3.5\\*.*')
                               #返回 C:\\python 3.5 文件夹中所有文件列表
>>>glob.iglob('C:\\python 3.5\\*.*')
                               #返回包含指定文件夹中所有文件的生成器对象
>>>glob.glob('tools\\**\\*.txt',recursive=True)
                               #递归查找 tools 文件夹中所有.txt 文件
>>>glob._rlistdir('.')         #递归遍历当前文件夹中所有文件,返回生成器对象
>>>glob.glob1('dlls','*.pyd')  #返回指定文件夹中指定类型的文件列表
>>>for i in glob.glob2('tools','**'):
    print(i)                   #递归遍历 tools 文件夹所有文件
```

10.4.2 fnmatch 模块

Python 标准库 fnmatch 提供了文件名的检查功能,支持通配符的使用,其中通配符 * 可以匹配任意字符,"?"可以匹配任何单个字符,[seq]可以匹配 seq 中的任何字符,[!seq]可以匹配任何不属于 seq 的字符。

标准库 fnmatch 提供的 fnmatch(filename, pattern)函数用来检查文件名 filename 是否与模式 pattern 相匹配,返回 True 或 False;fnmatchcase(filename, pattern)完成相似的功能,区别在于该函数区分大小写。

```
>>>import fnmatch
>>>fnmatch.fnmatch(r'C:\windows\notepad.exe','*.exe')
True
>>>fnmatch.fnmatch(r'C:\windows\notepad.exe','???????.exe')
False
>>>fnmatch.fnmatch(r'notepad.exe','*.exe')
True
>>>fnmatch.fnmatch(r'notepad.exe','???????.exe')
True
>>>fnmatch.fnmatchcase(r'notepad.exe','???????.eXe')
False
>>>fnmatch.fnmatchcase(r'notepad.exe','???????.exe')
True
```

标准库 fnmatch 提供的 filter(names, pattern)函数用来返回 names 中符合 pattern 的那部分元素构成的列表,等价于[n for n in names if fnmatch(n, pattern)],但效率更高。

```
>>>fnmatch.filter(os.listdir('.'),'*.py')      #当前文件夹中的所有.py文件
```

10.5　精彩案例赏析

示例 10-1　把指定文件夹中的所有文件名批量随机化,保持文件类型不变。

```
from string import ascii_letters
from os import listdir,rename
from os.path import splitext,join
from random import choice,randint

def randomFilename(directory):
    for fn in listdir(directory):
        #切分,得到文件名和扩展名
        name,ext =splitext(fn)
        n =randint(5,20)
        #生成随机字符串作为新文件名
        newName =''.join((choice(ascii_letters) for i in range(n)))
        #修改文件名
        rename(join(directory,fn),join(directory,newName +ext))

randomFilename('C:\\test')
```

示例 10-2　计算文件的 CRC32 值和 MD5 值。

```
import sys
import zlib
import hashlib
import os.path

def crc32md5(filename):
    if os.path.isfile(filename):
        with open(filename,'rb') as fp:
            contents =fp.read()
        return (zlib.crc32(contents),hashlib.md5(contents).hexdigest())
    else:
        print('file not exists')

print(crc32md5('tttt.py'))
```

示例 10-3　编写程序,进行文件夹增量备份。

程序功能与用法:指定源文件夹与目标文件夹,自动检测自上次备份以来源文件夹中内容的改变,包括修改的文件、新建的文件、新建的文件夹等,自动复制新增或修改过的文件到目标文件夹中,自上次备份以来没有修改过的文件将被忽略而不复制,从而实现增

量备份。本例属于系统运维的范畴。

```
import os
import filecmp
import shutil
import sys

def autoBackup(scrDir,dstDir):
    if ((not os.path.isdir(scrDir)) or (not os.path.isdir(dstDir)) or
      (os.path.abspath(scrDir)!=scrDir) or
      (os.path.abspath(dstDir)!=dstDir)):
        usage()
    for item in os.listdir(scrDir):
        scrItem=os.path.join(scrDir,item)
        dstItem=scrItem.replace(scrDir,dstDir)
        if os.path.isdir(scrItem):
            #创建新增的文件夹,保证目标文件夹的结构与原始文件夹一致
            if not os.path.exists(dstItem):
                os.makedirs(dstItem)
                print('make directory'+dstItem)
            #递归调用自身函数
            autoBackup(scrItem,dstItem)
        elif os.path.isfile(scrItem):
            #只复制新增或修改过的文件
            if ((not os.path.exists(dstItem)) or
                (not filecmp.cmp(scrItem,dstItem,shallow=False))):
                shutil.copyfile(scrItem,dstItem)
                print('file:'+scrItem+'==>'+dstItem)

def usage():
    print('scrDir and dstDir must be existing absolute path of certain directory')
    print('For example:{0} c:\\olddir c:\\newdir'.format(sys.argv[0]))
    sys.exit(0)

if __name__=='__main__':
    if len(sys.argv)!=3:
        usage()
    scrDir,dstDir=sys.argv[1],sys.argv[2]
    autoBackup(scrDir,dstDir)
```

示例 10-4 编写程序,统计指定文件夹大小以及文件和子文件夹数量。本例也属于系统运维范畴,可用于磁盘配额的计算,例如 E-mail、博客、FTP、快盘等系统中每个账号所占空间大小的统计。

```
import os
```



```
totalSize=0
fileNum=0
dirNum=0

def visitDir(path):
    global totalSize
    global fileNum
    global dirNum
    for lists in os.listdir(path):
        sub_path=os.path.join(path,lists)
        if os.path.isfile(sub_path):
            fileNum=fileNum+1                                  #统计文件数量
            totalSize=totalSize+os.path.getsize(sub_path)      #统计文件总大小
        elif os.path.isdir(sub_path):
            dirNum=dirNum+1                                    #统计文件夹数量
            visitDir(sub_path)                                 #递归遍历子文件夹

def main(path):
    if not os.path.isdir(path):
        print('Error:"',path,'" is not a directory or does not exist.')
        return
    visitDir(path)

def sizeConvert(size):                                         #单位换算
    K,M,G=1024,1024**2,1024**3
    if size >=G:
        return str(size/G)+'G Bytes'
    elif size >=M:
        return str(size/M)+'M Bytes'
    elif size >=K:
        return str(size/K)+'K Bytes'
    else:
        return str(size)+'Bytes'

def output(path):
    print('The total size of '+path+' is:'+sizeConvert(totalSize)+'('+str(totalSize)+' Bytes)')
    print('The total number of files in '+path+' is:',fileNum)
    print('The total number of directories in '+path+' is:',dirNum)

if __name__=='__main__':
    path=r'd:\idapro6.5plus'
    main(path)
    output(path)
```

示例 10-5　编写程序，递归删除指定文件夹中指定类型的文件和大小为 0 的文件。

```
from os.path import isdir,join,splitext
from os import remove,listdir,chmod,stat

filetypes=('.tmp','.log','.obj','.txt')                    #指定要删除的文件类型

def delCertainFiles(directory):
    if not isdir(directory):
        return
    for filename in listdir(directory):
        temp=join(directory,filename)
        if isdir(temp):
            delCertainFiles(temp)                          #递归调用
        elif splitext(temp)[1] in filetypes or stat(temp).st_size==0:
            chmod(temp,0o777)                              #修改文件属性,获取删除权限
            remove(temp)                                   #删除文件
            print(temp,' deleted…')

delCertainFiles(r'C:\test')
```

第 11 章 代码质量保障：异常处理结构、程序调试与测试

程序出错是一件非常难避免的事情。再厉害的程序员也无法提前预见代码运行时可能会遇到的所有情况，几乎每个程序员都被用户说过“你编的那个软件不好用啊”，而程序员经过反复检查以后发现问题的原因是用户操作不规范或者输入了错误类型的数据，于是一边修改代码加强类型检查一边抱怨用户不按套路出牌。其实呢，作者个人认为这样的问题的根源还是在程序员而不在用户，程序员编写代码时有义务也有必要考虑这些特殊情况，因为大多时候恰恰是少数特殊情况影响了整个系统的美感和开发人员的成就感（二八定律）。虽然大部分软件在发布前一般都经过了严格的测试，然而再充分的测试也很难枚举所有可能出现的情况，这时候异常处理结构则是避免特殊情况下软件崩溃的利器。

异常是指程序运行时引发的错误，引发错误的原因有很多，例如除零、下标越界、文件不存在、网络异常等。如果这些错误得不到正确的处理将会导致程序崩溃并终止运行，合理地使用异常处理结构可以使得程序更加健壮，具有更高的容错性，不会因为用户不小心的错误输入而造成程序崩溃，也可以使用异常处理结构为用户提供更加友好的提示。另外，有效的软件测试方法能够在软件发布之前发现尽可能多的 bug，而软件发布之后再出现错误时是否能够调试程序并快速定位和解决存在的问题则是程序员综合水平和能力的重要体现。

11.1 异常处理结构

11.1.1 异常的概念与表现形式

当程序执行过程中出现错误时 Python 会自动引发异常，程序员也可以通过 raise 语句显式地引发异常。异常处理是因为程序执行过程中由于输入不合法导致程序出错而在正常控制流之外采取的行为。严格来说，语法错误和逻辑错误不属于异常，但有些语法错误往往会导致异常，例如，由于大小写拼写错误而试图访问不存在的对象，或者试图访问不存在的文件，等。当 Python 检测到一个错误时，解释器就会指出当前程序流已经无法再继续执行下去，这时候就出现了异常。代码一旦抛出异常而得不到及时的处理，整个程序就会崩溃而提前结束，而合理地使用异常处理结构可以使得程序更加健壮，具有更高的容错性，不会因为用户不小心的错误输入而造成程序终止，也可以使用异常处理结构为用

户提供更加友好的提示。

在前面的章节中已经多次出现过异常,想必大家已经有了初步的了解。下面是几种比较常见的异常的表现形式:

```
>>>2 / 0                                        #除 0 错误
ZeroDivisionError: division by zero
>>>'a' +2                              #操作数类型不支持,略去异常的详细信息
TypeError: Can't convert 'int' object to str implicitly
>>>{3,4,5} *8                                   #操作数类型不支持
TypeError: unsupported operand type(s) for * : 'set' and 'int'
>>>print(testStr)                               #变量名不存在
NameError: name 'testStr' is not defined
>>>fp=open(r'D:\test.data','rb')                #文件不存在
FileNotFoundError: [Errno 2] No such file or directory: 'D:\\test.data'
>>>len(3)                                       #参数类型不匹配
TypeError: object of type 'int' has no len()
>>>list(3)                                      #参数类型不匹配
TypeError: 'int' object is not iterable
```

11.1.2 Python 内置异常类层次结构

下面全面展示了 Python 内置异常类的继承层次,其中 BaseException 是所有内置异常类的基类。在使用异常处理结构捕获和处理异常时,应尽量具体一点,最好是明确指定要捕获和处理哪一类异常。建议先尝试捕获派生类,然后再捕获基类,应尽量避免直接捕获 Exception 或 BaseException。

```
BaseException
 +--SystemExit
 +--KeyboardInterrupt
 +--GeneratorExit
 +--Exception
      +--StopIteration
      +--ArithmeticError
      |    +--FloatingPointError
      |    +--OverflowError
      |    +--ZeroDivisionError
      +--AssertionError
      +--AttributeError
      +--BufferError
      +--EOFError
      +--ImportError
      +--LookupError
      |    +--IndexError
      |    +--KeyError
```



```
+--MemoryError
+--NameError
|    +--UnboundLocalError
+--OSError
|    +--BlockingIOError
|    +--ChildProcessError
|    +--ConnectionError
|    |    +--BrokenPipeError
|    |    +--ConnectionAbortedError
|    |    +--ConnectionRefusedError
|    |    +--ConnectionResetError
|    +--FileExistsError
|    +--FileNotFoundError
|    +--InterruptedError
|    +--IsADirectoryError
|    +--NotADirectoryError
|    +--PermissionError
|    +--ProcessLookupError
|    +--TimeoutError
+--ReferenceError
+--RuntimeError
|    +--NotImplementedError
+--SyntaxError
|    +--IndentationError
|         +--TabError
+--SystemError
+--TypeError
+--ValueError
|    +--UnicodeError
|         +--UnicodeDecodeError
|         +--UnicodeEncodeError
|         +--UnicodeTranslateError
+--Warning
     +--DeprecationWarning
     +--PendingDeprecationWarning
     +--RuntimeWarning
     +--SyntaxWarning
     +--UserWarning
     +--FutureWarning
     +--ImportWarning
     +--UnicodeWarning
     +--BytesWarning
     +--ResourceWarning
```

11.1.3 异常处理结构

Python 提供了多种不同形式的异常处理结构,基本思路都是一致的:先尝试运行代码,如果没有问题就正常执行,如果发生了错误就尝试着去捕获和处理,最后实在没办法了才崩溃。从这个角度来看,不同形式的异常处理结构也属于选择结构的变形。

1. try…except…

Python 异常处理结构中最简单的形式是 try…except…结构,类似于单分支选择结构。其中 try 子句中的代码块包含可能会引发异常的语句,而 except 子句则用来捕捉相应的异常。如果 try 子句中的代码引发异常并被 except 子句捕捉,就执行 except 子句的代码块;如果 try 中的代码块没有出现异常就继续往下执行异常处理结构后面的代码;如果出现异常但没有被 except 捕获,继续往外层抛出;如果所有层都没有捕获并处理该异常,程序崩溃并将该异常呈现给最终用户。该结构语法如下:

```
try:
    #可能会引发异常的代码,先执行一下试试
except Exception[ as reason]:
    #如果 try中的代码抛出异常并被 except 捕捉,就执行这里的代码
```

下面的代码用来接收用户输入,并且要求用户必须输入整数,不接收其他类型的输入。

```
>>>while True:
    x=input('Please input:')
    try:
        x=int(x)
        print('You have input {0}'.format(x))
        break
    except Exception as e:
        print('Error.')

Please input:234c
Error.
Please input:5
You have input 5
```

2. try…except…else…

带有 else 子句的异常处理结构可以看作是一种特殊的双分支选择结构,如果 try 中的代码抛出了异常并且被 except 语句捕捉则执行相应的异常处理代码,这种情况下就不会执行 else 中的代码;如果 try 中的代码没有引发异常,则执行 else 块的代码。该结构的语法如下:



```
try:
    #可能会引发异常的代码
except Exception [ as reason]:
    #用来处理异常的代码
else:
    #如果 try 子句中的代码没有引发异常，就继续执行这里的代码
```

例如，前面要求用户必须输入整数的代码也可以像这样写，并且这是推荐的写法。也就是说，不要把太多代码放在 try 中，而是应该只放真的可能会引发异常的代码。

```
>>>while True:
    x=input('Please input:')
    try:
        x=int(x)
    except Exception as e:
        print('Error.')
    else:
        print('You have input {0}'.format(x))
        break

Please input:888c
Error.
Please input:888
You have input 888
```

3. try…except…finally…

在这种结构中，无论 try 中的代码是否发生异常，也不管抛出的异常有没有被 except 语句捕获，**finally 子句中的代码总是会得到执行**。因此，finally 中的代码常用来做一些清理工作，例如释放 try 子句中代码申请的资源。该结构语法为

```
try:
    #可能会引发异常的代码
except Exception [ as reason]:
    #处理异常的代码
finally:
    #无论 try 子句中的代码是否引发异常，都会执行这里的代码
```

例如下面的代码，不论是否发生异常，finally 子句中的代码总是被执行。

```
>>>def div(a,b):
    try:
        print(a/b)
    except ZeroDivisionError:
        print('The second parameter cannot be 0.')
    finally:
```



```
        print(-1)

>>>div(3,5)
0.6
-1
>>>div(3,0)
The second parameter cannot be 0.
-1
```

如果 try 子句中的异常没有被 except 语句捕捉和处理,或者 except 子句或 else 子句中的代码抛出了异常,那么这些异常将会在 finally 子句执行完后再次抛出。

```
>>>def div(a,b):
    try:
        print(a/b)
    except ZeroDivisionError:
        print('The second parameter cannot be 0.')
    finally:
        print(-1)

>>>div('3',5)
-1
(此处略去异常的详细信息)
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

需要注意的是,异常处理结构不是万能的,并不是采用了异常处理结构就万事大吉,finally 子句中的代码也可能会引发异常。下面代码的本意是使用 finally 子句来避免文件对象没有关闭的情况发生,但是由于指定的文件不存在而导致打开失败,结果在 finally 子句中关闭文件时引发了异常,因为这时并不存在文件对象 f1。

```
>>>try:
    f1 =open('test1.txt','r')       #文件不存在,抛出异常,不会创建文件对象 f1
    line =f1.readline()             #后面的代码不会被执行
    print(line)
except SyntaxError:                 #这个 except 并不能捕捉上面的异常
    print('Sth wrong')
finally:
    f1.close()                      #f1 不存在,再次引发异常

FileNotFoundError: [Errno 2] No such file or directory: 'test1.txt'
During handling of the above exception,another exception occurred:
NameError: name 'f1' is not defined
```

如果在函数中使用异常处理结构,尽量不要在 finally 子句中使用 return 语句,以免发生非常难以发现的逻辑错误。例如下面的代码,不管参数是否符合函数要求,调用函数时

都得到了同样的错误信息，因为 finally 子句中的代码总是会执行的。

```
>>>def div(a,b):
    try:
        return a/b
    except ZeroDivisionError:
        return 'The second parameter cannot be 0.'
    finally:
        return 'Error'

>>>div(3,5)
'Error'
>>>div('3',5)
'Error'
>>>div(3,0)
'Error'
```

4. 可以捕捉多种异常的异常处理结构

在实际开发中，同一段代码可能会抛出多种异常，并且需要针对不同的异常类型进行相应的处理。为了支持多种异常的捕捉和处理，Python 提供了带有多个 except 的异常处理结构，一旦 try 子句中的代码抛出了异常，就按顺序依次检查与哪一个 except 子句匹配，如果某个 except 捕捉到了异常，其他的 except 子句将不会再尝试捕捉异常。该结构类似于多分支选择结构，语法格式为

```
try:
    #可能会引发异常的代码
except Exception1:
    #处理异常类型 1 的代码
except Exception2:
    #处理异常类型 2 的代码
except Exception3:
    #处理异常类型 3 的代码
 ⋮
```

下面的代码演示了这种异常处理结构的用法，连续运行 3 次并输入不同的数据，结果如下：

```
>>>try:
    x=float(input('请输入被除数：'))
    y=float(input('请输入除数：'))
    z=x/y
except ZeroDivisionError:
    print('除数不能为零')
except TypeError:
```

```
    print('被除数和除数应为数值类型')
except NameError:
    print('变量不存在')
else:
    print(x,'/',y,'=',z)

请输入被除数:30                    #第一次运行
请输入除数:5
30.0/5.0 =6.0

请输入被除数:30                    #第二次运行,略去重复代码
请输入除数:abc
ValueError: could not convert string to float: 'abc'

请输入被除数:30                    #第三次运行,略去重复代码
请输入除数:0
除数不能为零
```

在实际开发中,有时候可能会为几种不同的异常设计相同的异常处理代码(虽然这种情况很少)。为了减少代码量,Python 允许把多个异常类型放到一个元组中,然后使用一个 except 子句同时捕捉多种异常,并且共用同一段异常处理代码。

```
>>>try:
     x =float(input('请输入被除数: '))
     y =float(input('请输入除数: '))
     z =float(x)/y
except (ZeroDivisionError,TypeError,NameError):
     print('捕捉到了异常')
else:
     print(x,'/',y,'=',z)

请输入被除数:30
请输入除数:0
捕捉到了异常
```

5. 同时包含 else 子句、finally 子句和多个 except 子句的异常处理结构

Python 异常处理结构中可以同时包含 else 子句、多个 except 子句和 finally 子句。例如:

```
>>>def div(x,y):
    try:
        print(x/y)
    except ZeroDivisionError:
        print('ZeroDivisionError')
    except TypeError:
        print('TypeError')
```



```
    else:
        print('No Error')
    finally:
        print("executing finally clause")

>>>div(3,5)
0.6
No Error
executing finally clause
>>>div('3',5)
TypeError
executing finally clause
>>>div(3,0)
ZeroDivisionError
executing finally clause
```

11.1.4　断言与上下文管理语句

断言语句 assert 也是一种比较常用的技术，常用来在程序的某个位置确认某个条件必须满足。断言语句 assert 仅当脚本的__debug__属性值为 True 时有效，一般只在开发和测试阶段使用。当使用优化选项-O 或-OO 把 Python 程序编译为字节码文件时，assert 语句将被删除。

```
>>>a=3
>>>b=5
>>>assert a==b,'a must be equal to b'

AssertionError: a must be equal to b
>>>try:
    assert a==b,'a must be equal to b'
except AssertionError as reason:
    print('%s:%s'%(reason.__class__.__name__,reason))

AssertionError:a must be equal to b
```

上下文管理(context manager)语句 with 可以自动管理资源，不论因为什么原因(哪怕是代码引发了异常)跳出 with 块，总能保证文件被正确关闭，并且可以在代码块执行完毕后自动还原进入该代码块时的现场，常用于文件操作、数据库连接、网络通信连接和多线程、多进程同步等场合。具体用法请参考第 9 章和第 12 章的案例。

11.2　文档测试 doctest

Python 标准库 doctest 可以搜索程序中类似于交互式 Python 代码的文本片段，并运行这些交互式代码来验证是否符合预期结果和功能，常用于 Python 程序的模块测试。下面

的代码演示了 doctest 模块的用法，定义了一个函数，预期功能为可以对整数或实数相加，或连接 2 个字符串、列表、元素，或对两个集合求并集，并返回结果。

示例 11-1 使用 doctest 模块测试 Python 代码。

```
def add(value1,value2):
    #下面一对三引号之间是测试代码,doctest 会搜索这些代码并执行
    #并且根据执行结果与预期结果的匹配程度来测试代码是否正确
    '''return the addition of two numbers or the concatenation of two string/list/tuple
     >>>add(3,5)
    8
     >>>add(3.0,5.0)
    8.0
     >>>add([1,2],[3,4])
    [1,2,3,4]
     >>>add((1,),(2,3,4))
    (1,2,3,4)
     >>>add(1,[3])
    Traceback (most recent call last):
        ⋮
    TypeError: value1 and value2 must be of the same type
     >>>add(1,'2')
    Traceback (most recent call last):
        ⋮
    TypeError: value1 and value2 must be of the same type
     >>>add([1],(2,))
    Traceback (most recent call last):
        ⋮
    TypeError: value1 and value2 must be of the same type
     >>>add('1234',[1,2,3,4])
    Traceback (most recent call last):
        ⋮
    TypeError: value1 and value2 must be of the same type
     >>>add({1,2,3},{3,4,5})
    {1,2,3,4,5}
     >>>add({1:1},{2:2})
    Traceback (most recent call last):
        ⋮
    TypeError: value1 and value2 must be the type of int,float,str,list,tuple or set
    '''
    #下面是正式的功能代码
    if type(value1) not in (int,float,str,list,tuple,set):
        raise TypeError('value1 and value2 must be the type of int,float,str,list,tuple or set')
    if type(value1) !=type(value2):
        raise TypeError('value1 and value2 must be of the same type')
    if type(value1) ==set:
        return value1 | value2
```

```
    else:
        return value1 +value2

if __name__=="__main__":
    import doctest
    doctest.testmod()
    print(add(3,5))
```

把上面的代码保存成 Python 程序文件 doctest_demo.py，在 IDE 中直接运行，如果函数功能完全符合预期的功能要求就会输出正确的结果，如果有不符合预期结果的代码就会给出相应的提示。在命令提示符环境中使用带-v 参数的方式执行，可以看到详细的测试过程，如图 11-1 所示。

```
命令提示符

C:\Python35>python test.py -v
Trying:
    add(3, 5)
Expecting:
    8
ok
Trying:
    add(3.0, 5.0)
Expecting:
    8.0
ok
Trying:
    add([1,2], [3, 4])
Expecting:
    [1, 2, 3, 4]
ok
Trying:
    add((1,), (2, 3, 4))
Expecting:
    (1, 2, 3, 4)
ok
Trying:
    add(1, [3])
Expecting:
    Traceback (most recent call last):
        ...
    TypeError: value1 and value2 must be of the same type
ok
Trying:
    add(1, '2')
Expecting:
    Traceback (most recent call last):
        ...
    TypeError: value1 and value2 must be of the same type
ok
Trying:
    add([1], (2,))
Expecting:
    Traceback (most recent call last):
        ...
    TypeError: value1 and value2 must be of the same type
ok
Trying:
    add('1234', [1,2,3,4])
Expecting:
    Traceback (most recent call last):
        ...
    TypeError: value1 and value2 must be of the same type
ok
Trying:
```

图 11-1　doctest 测试过程示意图

11.3　单元测试 unittest

软件测试对于保证软件质量非常重要，尤其是升级过程中不应影响系统原来的用法，是未来重构代码的信心保证。一般来说稍微有些规模的软件公司都有专门的测试团队来保证软件质量，但作为程序员，首先应该保证自己编写的代码准确无误地实现了预定功能。

软件测试方法有很多，从软件工程角度来讲，可以分为白盒测试和黑盒测试两大类。其中，白盒测试主要通过阅读程序源代码来判断是否符合功能要求，对于复杂的业务逻辑白盒测试难度非常大，一般以黑盒测试为主，白盒测试为辅。黑盒测试不关心模块的内部实现方式，只关心其功能是否正确，通过精心设计一些测试用例来检验模块的输入和输出是否正确，最终判断是否符合预定的功能要求。

单元测试是保证模块质量的重要手段之一，通过单元测试来管理设计好的测试用例，不仅可以避免测试过程中人工反复输入可能引入的错误，还可以重复利用设计好的测试用例，具有很好的可扩展性，大幅度缩短代码的测试时间。Python 标准库 unittest 提供了大量用于单元测试的类和方法，其中最常用的是 TestCase 类，其常用方法如表 11-1 所示。

表 11-1　TestCase 类的常用方法

方法名称	功 能 说 明	方法名称	功 能 说 明
assertEqual(a,b)	a == b	assertNotEqual(a,b)	a != b
assertTrue(x)	bool(x) is True	assertFalse(x)	bool(x) is False
assertIs(a,b)	a is b	assertIsNot(a,b)	a is not b
assertIsNone(x)	x is None	assertIsNotNone(x)	x is not None
assertIn(a,b)	a in b	assertNotIn(a,b)	a not in b
assertIsInstance(a,b)	isinstance(a,b)	assertNotIsInstance(a,b)	not isinstance(a,b)
assertAlmostEqual(a,b)	round(a-b,7) == 0	assertNotAlmostEqual(a,b)	round(a-b,7) != 0
assertGreater(a,b)	a > b	assertGreaterEqual(a,b)	a >= b
assertLess(a,b)	a < b	assertLessEqual(a,b)	a <= b
assertRegex(s,r)	r.search(s)	assertNotRegex(s,r)	not r.search(s)
setUp()	每项测试开始之前自动调用该函数	tearDown()	每项测试完成之后自动调用该函数

其中，setUp()和 tearDown()这两个方法比较特殊，分别在每个测试之前和之后自动调用，常用来执行数据库连接的创建与关闭、文件的打开与关闭等操作，避免编写过多的重复代码。

示例 11-2　编写单元测试程序。

以第 6 章自定义栈的代码为例，演示如何利用 unittest 库对 Stack 类中入栈、出栈、改

变大小以及满/空测试等方法进行测试，并将测试结果写入文件 test_Stack_result. txt。

```
from myStack import Stack
#Python 单元测试标准库
import unittest

class TestStack(unittest.TestCase):
    def setUp(self):
        #测试之前以追加模式打开指定文件
        self.fp=open('D:\\test_Stack_result.txt','a+')

    def tearDown(self):
        #测试结束后关闭文件
        self.fp.close()

    def test_isEmpty(self):
        try:
            s=Stack()
            #确保函数返回结果为 True
            self.assertTrue(s.isEmpty())
            self.fp.write('isEmpty passed\n')
        except Exception as e:
            self.fp.write('isEmpty failed\n')

    def test_clear(self):
        try:
            s=Stack(5)
            for i in ['a','b','c']:
                s.push(i)
            #测试清空栈操作是否工作正常
            s.clear()
            self.assertTrue(s.isEmpty())
            self.fp.write('clear passed\n')
        except Exception as e:
            self.fp.write('clear failed\n')

    def test_isFull(self):
        try:
            s=Stack(3)
            s.push(1)
            s.push(2)
            s.push(3)
            self.assertTrue(s.isFull())
            self.fp.write('isFull passed\n')
```

```
            except Exception as e:
                self.fp.write('isFull failed\n')

        def test_pushpop(self):
            try:
                s =Stack()
                s.push(3)
                #确保入栈后立刻出栈得到原来的元素
                self.assertEqual(s.pop(),3)
                s.push('a')
                self.assertEqual(s.pop(),'a')
                self.fp.write('push and pop passed\n')
            except Exception as e:
                self.fp.write('push or pop failed\n')

        def test_setSize(self):
            try:
                s =Stack(8)
                for i in range(8):
                    s.push(i)
                self.assertTrue(s.isFull())
                #测试扩大栈空间是否正常工作
                s.setSize(9)
                s.push(8)
                self.assertTrue(s.isFull())
                self.assertEqual(s.pop(),8)
                #测试缩小栈空间是否正常工作
                s.setSize(4)
                self.assertEqual(s._size,9)
                self.fp.write('setSize passed\n')
            except Exception as e:
                self.fp.write('setSize failed\n')

if __name__=='__main__':
    unittest.main()
```

在 Eclipse + PyDev 环境中,Python 程序有 Python Run 和 Python unittest 两种运行方式,前者只运行"if __name__ = = '__main__':"这一行所限定的代码块,而后者会执行所有的测试代码,也就是继承自 unittest. TestCase 的类中所有以 test 开头的方法。

在进行单元测试时应注意:①测试用例的设计应该是完备的,应保证覆盖尽可能多的情况,尤其是要覆盖边界条件,对目标模块的功能进行充分测试,避免漏测;②测试用例以及测试代码本身也可能会存在 bug,通过测试并不代表目标代码没有错误,但是一般而言,不能通过测试的模块代码是存在问题的;③再好的测试方法和测试用例也无法保



证能够发现所有错误，必须通过不停改进和综合多种测试方法并且精心设计测试用例来发现尽可能多的潜在问题；④除了功能测试，还应对程序进行性能测试与安全性测试，甚至还需要进行规范性测试以保证代码可读性和可维护性。

11.4　覆盖测试

覆盖测试通过代码分析工具和跟踪钩子来判断哪些代码可执行以及哪些代码被执行了，是对单元测试的有效补充，可以用来判断测试的有效性。

Python 扩展库 coverage 可以实现对 Python 代码的覆盖测试，使用 pip 工具安装之后，可以使用命令 coverage run file. py 对 Python 程序 file. py 进行覆盖测试，然后使用命令 coverage report 直接查看测试报告，或者使用命令 coverage html 生成 HTML 文件的测试报告，这些 HTML 文件自动保存在 htmlcov 文件夹中。可以使用命令 coverage help 查看 coverage 支持的所有命令。

例如，有下面的代码用来判断一个整数是否为素数：

```
from random import randint

def isPrime(n):
    for i in range(2,int(n**0.5)+1):
        if n%i==0:
            return 'No'
    else:
        return 'Yes'

n=randint(3,2000)
print(n,':',isPrime(n))
```

把上面的代码保存为 isPrime. py，然后在命令提示符环境中首先执行命令 coverage run isPrime. py 测试，再执行命令 coverage report 查看测试报告。-m 选项用来显示没有被执行到的代码行号，可以使用命令 coverage report -h 查看更多选项。

```
C:\Python 3.5>coverage run isPrime.py
1862 : No

C:\Python 3.5>coverage report
Name         Stmts  Miss  Cover
--------------------------------
isprime.py      8      1    88%

C:\Python 3.5>coverage report -m
Name         Stmts  Miss  Cover  Missing
-----------------------------------------
isprime.py      8      1    88%     8
```

另外，扩展库 coverage 还提供了编程接口支持代码覆盖测试。例如，把上面的素数判断程序修改为下面的代码并执行，会自动生成测试报告。

```
import coverage
from random import randint

cov=coverage.Coverage()
cov.start()

def isPrime(n):
    for i in range(2,int(n**0.5)+2):
        if n%i==0:
            return 'No'
    else:
        return 'Yes'

n=randint(3,2000)
print(n,':',isPrime(n))

cov.stop()
cov.save()
cov.html_report()
```

11.5 软件性能测试

在本书前面章节中多次演示过使用 Python 标准库 time 和 timeit 提供的函数来测试代码运行时间。除了前面介绍的用法，还可以使用下面的方法来测试代码运行时间：

```
from time import time

class Timer(object):
    def __enter__(self):
        self.start=time()
        return self

    def __exit__(self, *args):
        self.end=time()
        self.seconds=self.end-self.start

def isPrime(n):
    if n==2:
        return True
    for i in range(2,int(n**0.5)+2):
        if n%i==0:
```




```
                return False
        return True

with Timer() as t:
        for i in range(1000):
                isPrime(9999999999999999999999)
print(t.seconds)
```

运行上面的程序，会输出 isPrime( ) 函数运行 1000 次所需要的时间。另外，在很多时候，除了要测试代码运行所需要的时间，还需要检测代码运行过程中的内存占用情况，这时候需要使用 pip 安装 Python 扩展库 memory_profiler，然后编写下面的代码：

```
from memory_ profiler import profile

@profile                                #修饰器
def isPrime(n):
        if n==2:
                return True
        for i in range(2,int(n**0.5)+2):
                if n% i==0:
                        return False
        return True

isPrime(99999999999999999999999)
```

运行上面的程序，会得到下面的输出，从中可以清楚地看到代码对内存的使用情况：

```
Line #    Mem usage    Increment    Line Contents
================================================
     3     33.9 MiB     0.0 MiB    @profile
     4                               def isPrime(n):
     5     33.9 MiB     0.0 MiB        if n==2:
     6                                        return True
     7     33.9 MiB     0.0 MiB        for i in range(2,int(n* * 0.5)+2):
     8     33.9 MiB     0.0 MiB            if n% i==0:
     9     33.9 MiB     0.0 MiB                return False
    10                                    return True
```

利用 Python 标准库 cProfile 也可以对 Python 程序进行测试，并给出统计信息，包括代码的执行次数、运行时间等，为代码测试和性能优化提供有力依据。

```
>>>def fac(n):
    assert isinstance(n,int) and n>=0
    if n in (0,1):
        return 1
    time.sleep(0.1)
```

```
    return n * fac(n-1)
>>>import time
>>>cProfile.run('fac(30)')
         92 function calls (63 primitive calls) in 2.899 seconds

  Ordered by: standard name

  ncalls  tottime  percall  cumtime  percall filename:lineno(function)
  30/1    0.001    0.000    2.899    2.899   <pyshell#4>:1(fac)
     1    0.000    0.000    2.899    2.899   <string>:1(<module>)
     1    0.000    0.000    2.899    2.899   {built-in method builtins.exec}
    30    0.000    0.000    0.000    0.000   {built-in method builtins
                                             .isinstance}
    29    2.899    0.100    2.899    0.100   {built-in method time.sleep}
     1    0.000    0.000    0.000    0.000   {method 'disable' of '_lsprof
                                             .Profiler' objects}
```

上面的测试结果中各部分含义如表 11-2 所示。

表 11-2　cProfile 测试结果各部分含义

名　称	含　　义
ncalls	调用次数
tottime	该函数执行所用的总时间，不包括调用子函数所用的时间
percall	该函数单次执行所用的时间
cumtime	该函数及其所有子函数执行所用总时间
percall	该函数及其所有子函数单次执行所用时间
filename:lineno(function)	函数或代码有关信息

11.6　代码调试

11.6.1　使用 IDLE 调试

当程序运行发生错误或者得到了非预期的结果时，是否能够熟练地对程序进行调试并快速定位和解决问题是体现程序员综合能力的重要标准之一。

使用 IDLE 的调试功能时，首先单击 IDLE 的菜单 Debug→Debugger 打开调试器窗口，然后打开并运行要调试的程序，最后切换到调试器窗口使用其中的控制按钮进行调试。图 11-2 为 IDLE 调试窗口及其功能简要介绍，可以使用调试按钮对程序进行单步执行，实时查看变量的当前值并跟踪其变化过程，对于理解程序内部工作原理和发现程序中存在的问题非常有帮助。





图 11-2　IDLE 调试器窗口

示例 11-3　使用 IDLE 调试 Python 程序。

假设有 Python 程序 demo.py，其功能为生成 1000 个随机字符（英语字母大小写或数字），然后查看某个字符的出现次数，代码如下：

```
import string
from random import choice

characters =string.ascii_letters +string.digits
selected=[choice(characters) for i in range(1000)]
ch =choice(selected)
print(ch,':',selected.count(ch))
```

然后使用 IDLE 对该程序进行单步调试（使用 Step 按钮），调试过程中的部分截图如图 11-3 和图 11-4 所示。可以发现，在调试过程中执行了很多不属于 demo.py 程序的代码，这是正常的，因为调用标准库函数时会自动进入标准库并执行其中的代码。如果不想进入和执行标准库代码，可以使用 Over 按钮。

11.6.2　使用 Eclipse + PyDev 进行代码调试

Eclipse + PyDev 提供了更加强大的代码调试功能，使用也更加方便和灵活。在代码窗口中某行左侧的标尺上右击，在弹出的菜单中使用 Add BreakPoint 在该行设置断点，然后使用菜单 Run→Debug As→Python Run，在弹出的窗口中选择 Yes 进入调试器界面，最后使用菜单或快捷键 F5（Step Into）、F6（Step Over）、F7（Step Return）或 F8（Resume）进行调试，并在相应的窗口中查看程序状态和变量的当前值。调试界面如图 11-5 所示。



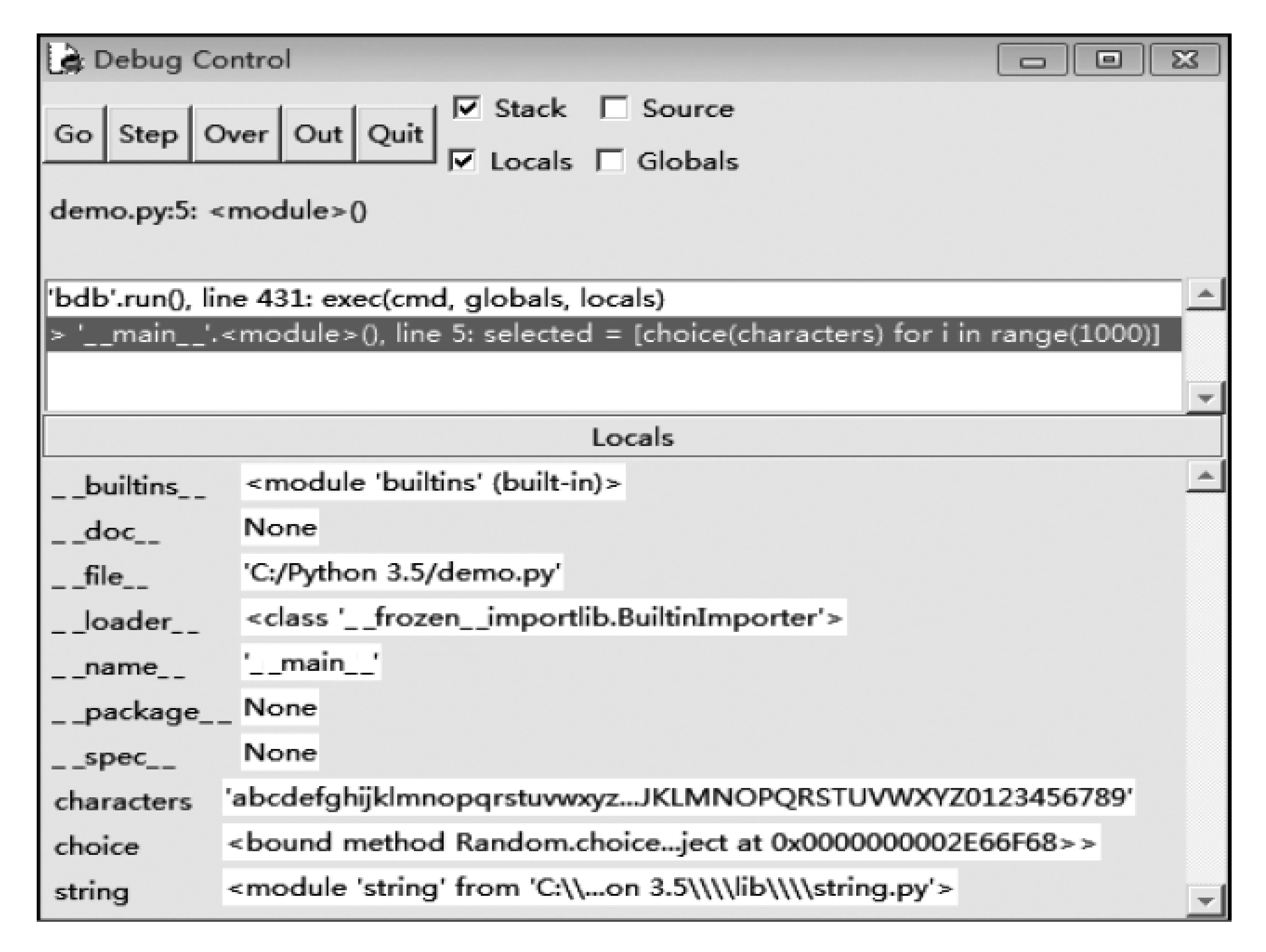


图 11-3 程序调试截图(一)

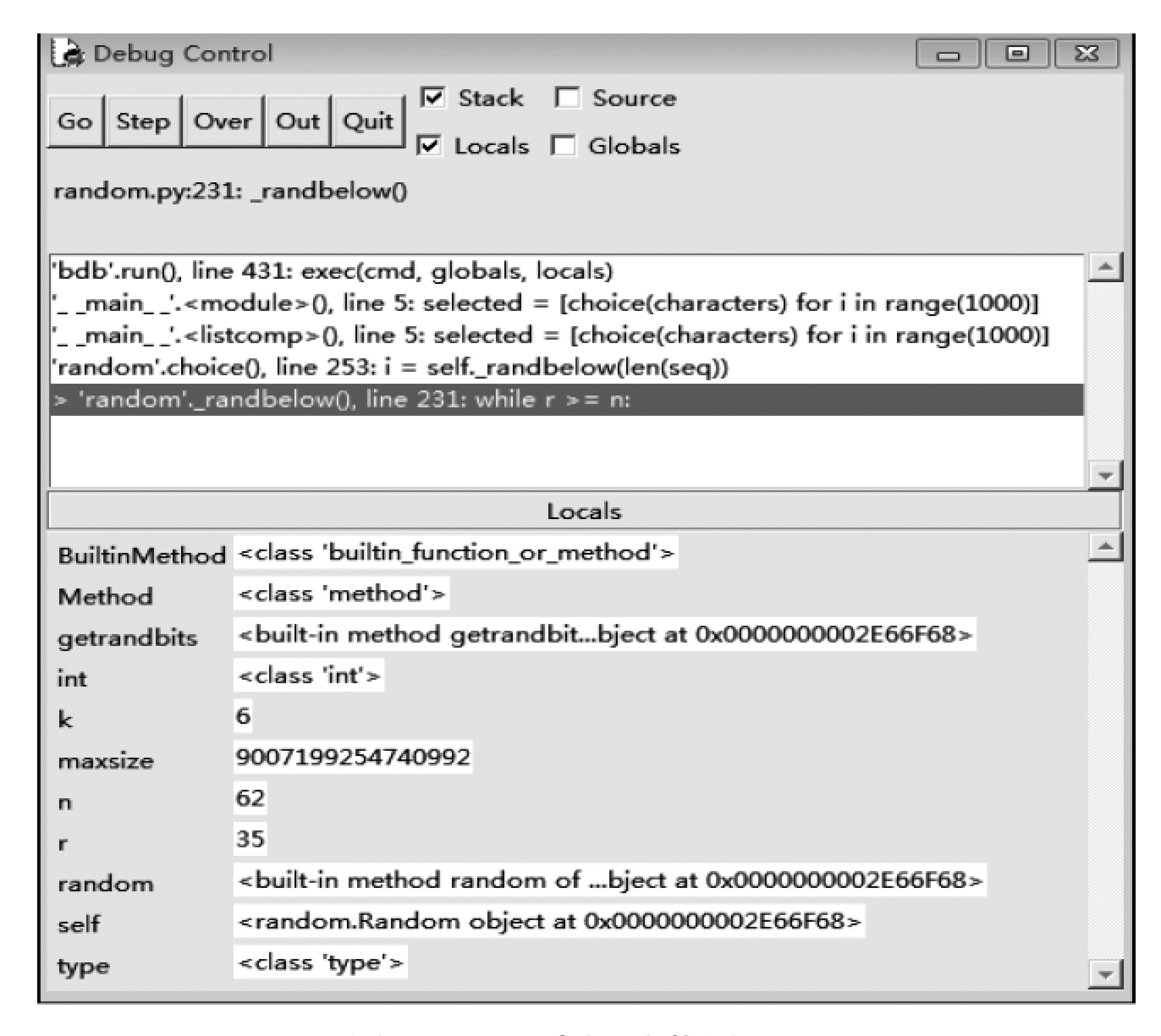


图 11-4 程序调试截图(二)



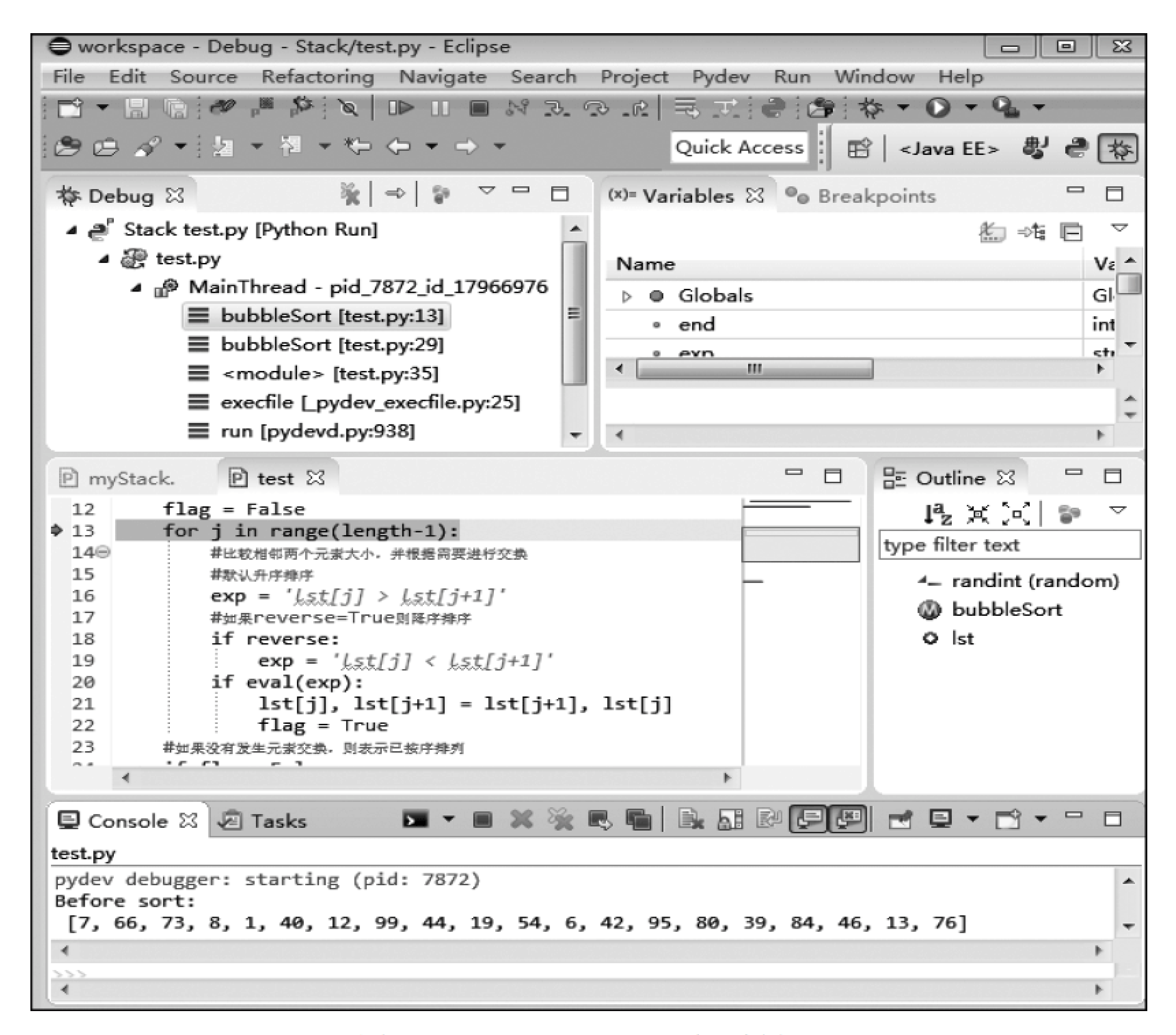


图 11-5　Eclipse + PyDev 调试界面

11.6.3　使用 pdb 调试

pdb 是 Python 自带的交互式源代码调试模块，其源文件为 pdb. py，感兴趣的读者可以在 Python 安装目录下找到该文件进行阅读并理解其工作原理。pdb 模块提供了代码调试所需要的绝大部分功能，包括设置/清除（条件）断点、启用/禁用断点、单步执行、查看栈帧、查看变量值、查看当前执行位置、列出源代码、执行任意 Python 代码或表达式等。pdb 还支持事后调试，可在程序控制下被调用，并且可以通过 pdb 和 cmd 接口对该调试器进行扩展。pdb 模块常用调试命令如表 11-3 所示。

使用 pdb 模块调试 Python 代码的形式常见的有 3 种：在交互模式下调试特定的代码块，在程序中显式插入断点，把 pdb 作为模块来调试程序。

（1）在交互模式下使用 pdb 模块提供的功能可以直接调试语句块、表达式、函数等多种脚本，常用的调试方法如下。

表 11-3　常用 pdb 调试命令

简写/完整命令	用法示例	解　　释
a(rgs)		显示当前函数中的参数
b(reak) [[filename:]lineno | function [, condition]]	b 173	在 173 行设置断点
	b function	在 function 函数第一条可执行语句位置设置断点
	b	不带参数则列出所有断点,包括每个断点的触发次数、当前忽略计数以及与之关联的条件
	b 175, condition	设置条件断点,仅当 condition 的值为 True 时该断点有效
cl(ear) [filename:lineno | bpnumber [bpnumber …]]	cl	清除所有断点
	cl file:line	删除指定文件中指定行的所有断点
	cl 3 5 9	删除第 3、5、9 个断点
condition bpnumber [condition]	condition 3 a < b	仅当 a < b 时 3 号断点有效
	condition 3	将 3 号断点设置为无条件断点
continue		继续运行至下一个断点或脚本结束
disable [bpnumber [bpnumber …]]	disable 3 5	禁用第 3、5 个断点,禁用后断点仍存在,可以再次被启用
d(own)		在栈跟踪器中向下移动一个栈帧
enable [bpnumber [bpnumber …]]	enable n	启用第 n 个断点
h(elp) [command]		查看 pdb 帮助
ignore bpnumber [count]		为断点设置忽略计数,count 默认值为 0。若某断点的忽略计数不为 0,则每次触发时自动减 1,当忽略计数为 0 时该断点处于活动状态
j(ump)	j 20	跳至第 20 行继续运行
l(ist) [first [, last]]	l	列出脚本清单,默认 11 行
	l m, n	列出从第 m 行到第 n 行之间的脚本代码
	l m	列出从第 m 行开始的 11 行代码
n(ext)		执行下一条语句,遇到函数时不进入其内部
p(rint)	p i	打印变量 i 的值
q(uit)		退出 pdb 调试环境
r(eturn)		一直运行至当前函数返回
tbreak		设置临时断点,该类型断点只被中断一次,触发后该断点自动删除
step		执行下一条语句,遇到函数时进入其内部
u(p)		在栈跟踪器中向上移动一个栈帧

续表

简写/完整命令	用法示例	解　　释
w(here)		查看当前栈帧
[!]statement		在 pdb 中执行语句，!与要执行的语句之间不需要空格，任何非 pdb 命令都被解释为 Python 语句并执行，甚至可以调用函数或修改当前上下文中变量的值
		直接回车则默认执行上一个命令

① pdb. run(statement[, globals[, locals]])：调试指定语句，可选参数 globals 和 locals 用来指定代码执行的环境，默认是__main__模块的字典。

② pdb. runeval(expression[, globals[, locals]])：返回表达式的值，可选参数 globals 和 locals 的含义与上面的 run()函数一样。

③ pdb. runcall(function[, argument,…])：调试指定函数。

④ pdb. post_mortem([traceback])：进入指定 traceback 对象的事后调试模式，如果没有指定 traceback 对象，则使用当前正在处理的一个异常。

例如，下面的代码演示了如何调试一个函数，其中“(Pdb)”为提示符，在后面输入并执行前面表 11-3 中介绍的命令即可。

```
>>>import pdb
>>>def demo():
    from random import randint
    x=[randint(1,10) for i in range(20)]        #随机生成 20 个介于 1~10 的整数
    m=max(x)                                    #最大数
    r=[index for index,value in enumerate(x) if value==m]
    print(r)                                    #输出最大数所在的下标

>>>pdb.runcall(demo)                            #调试函数
><pyshell#3>(2)demo()
(Pdb) n                                         #执行下一条语句
><pyshell#3>(3)demo()
(Pdb) n
><pyshell#3>(4)demo()
(Pdb) p x                                       #查看变量值
[9,5,8,9,7,8,10,3,8,5,2,8,8,4,9,10,8,7,5,1]
(Pdb) p m
*** NameError: name 'm' is not defined
(Pdb) n
><pyshell#3>(5)demo()
(Pdb) p m
10
(Pdb) r                                         #运行函数直至结束
[6,15]
--Return--
```

```
><pyshell#3>(6)demo()->None
(Pdb) l
[EOF]
(Pdb) p r
[6,15]
(Pdb) p m
10
(Pdb) q                                         #退出调试模式
```

(2) 在程序中嵌入断点来实现调试功能。

在程序中首先导入 pdb 模块,然后使用 pdb. set_trace( )在需要的位置设置断点。如果程序中存在这样插入的断点,那么在命令提示符环境下执行该程序或双击执行程序时将自动进行 pdb 调试模式,即使该程序当前不处于调试状态。例如,下面的程序 IsPrime. py:

```
import pdb

n=37
pdb.set_trace()
for i in range(2,n):
    if n%i==0:
        print('No')
        break
else:
    print('Yes')
```

由于使用 pdb 设置了断点,在 IDLE 中运行该程序时会自动打开调试模式,如图 11-6 所示。在命令提示符环境中运行该程序时也会自动进入 pdb 调试模式,如图 11-7 所示。

```
====================== RESTART: C:/Pytho
> c:\python35\isprime.py(5)<module>()
-> for i in range(2, n):
(Pdb) n
> c:\python35\isprime.py(6)<module>()
-> if n%i == 0:
(Pdb) n
> c:\python35\isprime.py(5)<module>()
-> for i in range(2, n):
(Pdb) p i
2
(Pdb) n
> c:\python35\isprime.py(6)<module>()
-> if n%i == 0:
(Pdb) n
> c:\python35\isprime.py(5)<module>()
-> for i in range(2, n):
(Pdb) n
> c:\python35\isprime.py(6)<module>()
-> if n%i == 0:
(Pdb) p i
4
(Pdb) a
(Pdb) l
  1         import pdb
  2
  3         n = 37
  4         pdb.set_trace()
  5         for i in range(2, n):
  6  ->         if n%i == 0:
  7                 print('No')
  8                 break
  9         else:
 10             print('Yes')
[EOF]
(Pdb)
```

图 11-6 运行程序自动进行 pdb 调试模式

```
C:\Python35>python IsPrime.py
> c:\python35\isprime.py(5)<module>()
-> for i in range(2,n):
(Pdb) p i
*** NameError: name 'i' is not defined
(Pdb) n
> c:\python35\isprime.py(6)<module>()
-> if n%i == 0:
(Pdb) p i
2
(Pdb) l
  1         import pdb
  2
  3         n = 37
  4         pdb.set_trace()
  5         for i in range(2,n):
  6  ->         if n%i == 0:
  7                 print('No')
  8                 break
  9         else:
 10             print('Yes')
[EOF]
(Pdb)
```

图 11-7 在命令提示符环境运行程序



（3）使用命令行调试程序。

在命令行提示符下执行“python -m pdb 脚本文件名”，可以直接进入调试环境，即使程序中并没有设置任何断点，也没有使用 pdb 的任何功能；当调试结束或程序正常结束以后，pdb 将重启该程序。例如，把上面的程序 IsPrime. py 中 pdb 模块的导入和断点插入函数都删除，然后在命令提示符环境中使用调试模式运行，如图 11-8 所示。

```
C:\Python35>python -m pdb IsPrime.py
> c:\python35\isprime.py(1)<module>()
-> n = 37
(Pdb) n
> c:\python35\isprime.py(2)<module>()
-> for i in range(2,n):
(Pdb) w
  c:\python35\lib\bdb.py(431)run()
-> exec(cmd, globals, locals)
  <string>(1)<module>()
> c:\python35\isprime.py(2)<module>()
-> for i in range(2,n):
(Pdb) n
> c:\python35\isprime.py(3)<module>()
-> if n%i == 0:
(Pdb) p i
2
(Pdb) p n
37
(Pdb)
```

图 11-8　以调试模式运行程序

第 12 章 多任务与并行处理：线程、进程、协程、分布式、GPU 加速

12.1 多线程编程

多线程技术的引入并不仅仅是为了提高处理速度和硬件资源的利用率，更重要的是可以提高系统的可扩展性（采用多线程技术编写的代码移植到多处理器平台上不需要改写就能立刻适应新的平台）和用户体验。虽然对于单核 CPU 计算机而言，使用多线程并不能提高任务完成速度，但有些场合必须要使用多线程技术，或者采用多线程技术可以让整个系统的设计更加人性化。例如，在执行一段代码的同时还可以接收和响应用户的键盘或鼠标事件以提高用户体验；Windows 操作系统的 Windows Indexing Services 创建了一个低优先级的线程，该线程定期被唤醒并对磁盘上的特定区域的文件内容进行索引以提高用户搜索速度；打开 Photoshop、3ds Max 这样的大型软件时需要加载很多模块和动态链接库，软件启动时间会比较长，可以使用一个线程来显示一个小动画来表示当前软件正在启动，当后台线程加载完所有的模块和库之后，结束该动画的播放并打开软件主界面；字处理软件可以使用一个优先级高的线程来接收用户键盘输入，而使用一些低优先级线程来进行拼写检查、语法检查、分页以及字数统计之类的功能并将结果显示在状态栏上，对于提高用户体验有重要帮助。图 12-1 展示了字处理软件 WPS 启动之后创建的部分线程，图 12-2 中列出了作者的计算机上某个时刻各进程拥有的线程数量。可见在系统运行过程中会同时存在大量的线程，这些线程的调用和同步必然需要有一定的算法和相应的机制，这也正是本节的内容。

12.1.1 线程概念与标准库 threading

磁盘上的应用程序文件被打开并执行时创建一个进程，但进程本身并不是可执行单元，从来不执行任何东西，主要用作线程和相关资源的容器。要使进程中的代码真正运行起来，必须拥有至少一个能够在这个环境中运行代码的执行单元，也就是线程。线程是操作系统调度的基本单位，负责执行包含在进程地址空间中的代码并访问其中的资源。当一个进程被创建时，操作系统会自动为之建立一个线程，通常称为主线程。一个进程可以包含多个线程，主线程根据需要再动态创建其他子线程，操作系统为每个线程保存单独的寄存器环境和单独的堆栈，但是它们共享进程的地址空间、对象句柄、代码、数据和其他资源。线程总是在某个进程的上下文中被创建、运行和结束，不可以脱离进程而独立存在，

但允许属于同一个进程中的多个线程之间进行数据共享和同步控制。一般来说，除主线程的生命周期与所属进程的生命周期一样之外，其他线程的生命周期都小于其所属进程的生命周期。

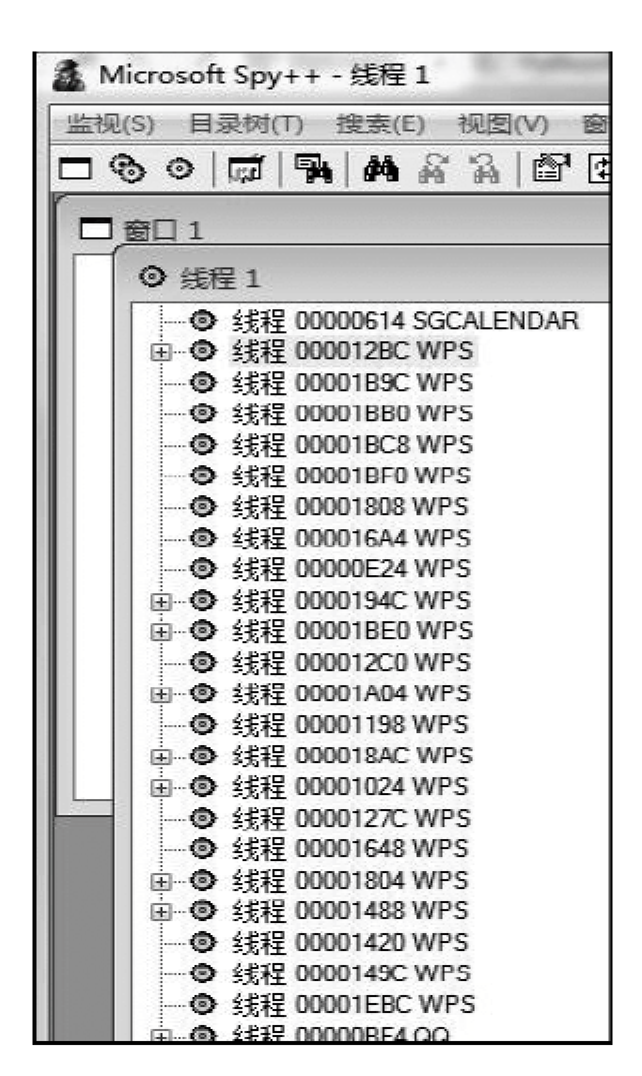


图 12-1　WPS 创建的部分线程

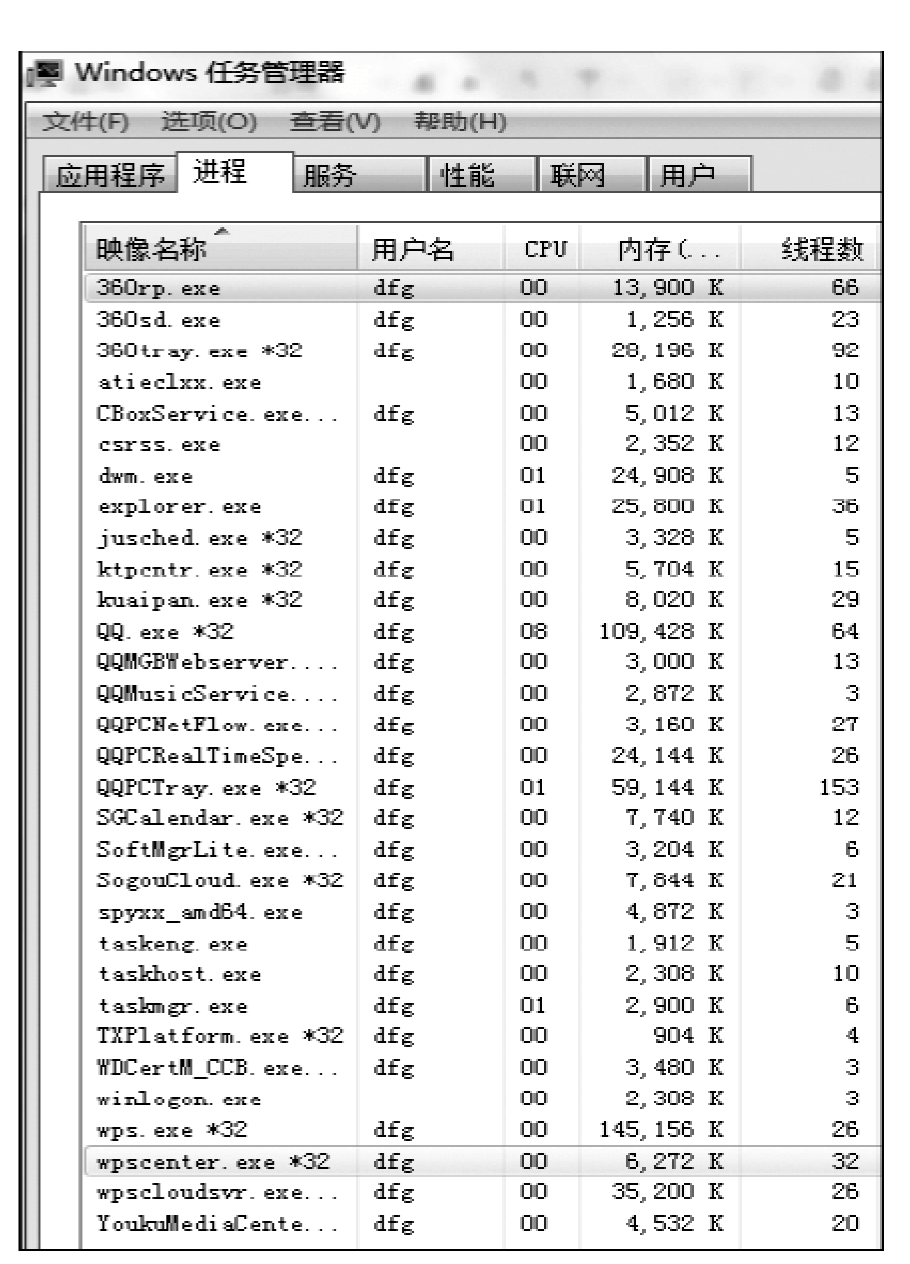


图 12-2　系统中每个进程的线程数量

标准库 threading 是 Python 支持多线程编程的重要模块，该模块是在底层模块 _thread 的基础上开发的更高层次的线程编程接口，提供了大量的方法和类来支持多线程编程，极大地方便了用户。标准库 threading 提供的常用方法如表 12-1 所示。

表 12-1　标准库 threading 提供的常用方法

方　　法	功 能 说 明
active_count()、activeCount()	返回当前处于 alive 状态的 Thread 对象数量
current_thread()、currentThread()	返回当前 Thread 对象
get_ident()	返回当前线程的线程标识符。线程标识符是一个非负整数，这个整数本身并没特殊含义，只是用来标识线程，可能会被循环利用
enumerate()	返回当前处于 alive 状态的所有 Thread 对象列表
local	线程局部数据类
main_thread()	返回主线程对象，即启动 Python 解释器的线程对象
stack_size([size])	返回创建线程时使用的栈的大小，如果指定 size 参数，则用来指定后续创建的线程使用的栈大小，size 必须是 0(表示使用系统默认值)或大于 32K 的正整数

续表

方　　法	功 能 说 明
setprofile(func)	设置之后每个线程启动之前都会把 func 函数传递给 sys. setprofile()
settrace(func)	设置之后每个线程启动之前都会把 func 函数传递给 sys. settrace()
TIMEOUT_MAX	线程同步获取锁时的最大允许等待时间
Thread	线程类,用于创建和管理线程
Event	事件类,用于线程同步
Condition	条件类,用于线程同步
Lock、RLock	锁类,用于线程同步
Semaphore、BoundedSemaphore	信号量类,用于线程同步
Timer	用于在指定时间之后调用一个函数

12.1.2 线程对象

在一个函数中直接调用另外一个函数时,当前函数中调用位置之后的代码会暂停执行,直到被调函数执行结束并正确返回后才能继续执行当前函数的代码。与直接调用函数不同,以创建并启动线程的方式来执行一个函数可以实现多个函数中的代码并发或同时运行。

标准库 threading 中的 Thread 类用来创建和管理线程对象,支持使用两种方法来创建线程:①直接使用 Thread 类实例化一个线程对象并传递一个可调用对象作为参数;②继承 Thread 类并在派生类中重写__init__()和 run()方法。创建了线程对象以后,可以调用其 start()方法来启动,该方法自动调用该类对象的 run()方法,此时该线程处于 alive 状态,直至线程的 run()方法运行结束。Thread 对象的主要成员如表 12-2 所示。

表 12-2　Thread 对象的主要成员

成　　员	说　　明
start()	自动调用 run()方法,启动线程,执行线程代码。每个线程只能启动一次
run()	线程代码,用来实现线程的功能与业务逻辑,可以在子类中重写该方法来自定义线程的行为
__init__(self, group = None, target = None, name = None, args = (), kwargs = None, verbose = None)	构造方法
name	读取或设置线程的名字
ident	线程标识,非 0 数字或 None(线程未被启动)
is_alive()、isAlive()	测试线程是否处于 alive 状态
daemon	布尔值,表示线程是否为守护线程
join(timeout = None)	等待线程结束或超时返回



另外，threading 还支持使用 Timer 类来创建定时启动的线程，调用线程的 start( ) 方法之后，线程会在指定的时间(单位是秒)之后再调用线程函数。

```
>>>def demo(v):
    print(v)
>>>t =threading.Timer(3,demo,args =(5,))          #创建线程
>>>t.start()                                       #启动线程,3s 之后调用 demo 函数
>>>t.cancel()                                      #如果仍在等待时间到达,则取消
```

1. join([timeout])

阻塞当前线程，等待被调线程结束或超时后再继续执行当前线程的后续代码，参数 timeout 用来指定最长等待时间，单位是秒。方法 join( ) 返回后再调用 isAlive( ) 方法，如果得到 True 则说明线程仍在运行并且 join( ) 方法是因为超时而返回的，如果 isAlive( ) 返回 False 则说明 join( ) 方法是因为线程运行结束而返回的。一个线程可以调用多次 join( ) 方法(如果线程已结束，join( ) 会立即返回)，但不允许对当前线程调用 join( )，否则会抛出异常。

示例 12-1　线程对象的 join( ) 方法。

```
from threading import Thread
import time

def func1(x,y):
    for i in range(x,y):
        print(i,end=' ')
    print()
    time.sleep(10)                                 #等待 10s

t1 =Thread(target =func1,args =(15,20))            #创建线程对象,args 是传递给函数的参数
t1.start()                                         #启动线程
t1.join(5)                                         #等待线程 t1 运行结束或等待 5s
t2 =Thread(target =func1,args =(5,10))
t2.start()
```

保存并运行上面的程序，首先输出 15 至 19 这 5 个整数，然后程序暂停 5s 以后又继续输出 5 至 9 这 5 个整数。如果把 t1. join(5) 这一行注释或删除之后再次运行，两个线程的输出将会重叠在一起，这是因为两个线程并发运行，而不是等待第一个结束以后再运行第二个。如果把 time. sleep(10) 这一行注释或删除再运行，会发现两个线程的输出之间没有时间间隔，这是因为线程对象的 join( ) 方法当线程运行结束或超时之后返回，虽然指定了超时时间为 5s，而实际上线程函数瞬间就执行结束了。

2. isAlive( )

这个方法用来测试线程是否处于运行状态，如果仍在运行则返回 True，如果尚未启

动或运行已结束则返回 False。

示例 12-2 线程状态检测。

```
from threading import Thread
import time

def func1():
    time.sleep(10)

t1 =Thread(target =func1)
print('t1:',t1.isAlive())                    #线程还没有运行,返回 False
t1.start()
print('t1:',t1.isAlive())                    #线程还在运行,返回 True
t1.join(5)                                   #join()方法因超时而结束
print('t1:',t1.isAlive())                    #线程还在运行,返回 True
t1.join()                                    #等待线程结束
print('t1:',t1.isAlive())                    #线程已结束,返回 False
```

上面程序的输出结果为

```
t1: False
t1: True
t1: True

t1: False
```

3. daemon 属性

在多线程编程中,如果子线程需要访问主线程中的资源(比如某个变量),当退出程序时主线程结束后这些资源将不再存在,子线程继续运行时会因为无法访问资源而引发异常导致崩溃。因此,需要有一种机制保证主线程结束时可以同时结束子线程,或者使得主线程等待子线程运行结束后再结束,线程的 daemon 属性恰好能够满足这样的要求。

在程序运行过程中有一个主线程,若在主线程中创建了子线程,当主线程结束时根据子线程 daemon 属性值的不同会发生下面的两种情况之一。

(1) 如果某个子线程的 daemon 属性为 False,主线程结束时会检测该子线程是否结束,如果该子线程还在运行,则主线程会等待它完成后再退出。

(2) 如果某个子线程的 daemon 属性为 True,主线程运行结束时不对这个子线程进行检查而直接退出,同时所有 daemon 值为 True 的子线程将随主线程一起结束,而不论是否运行完成。

属性 daemon 的值默认为 False,如果需要修改,必须在调用 start()方法启动线程之前进行设置。另外要注意的是,上面的描述并不适用于 IDLE 环境中的交互模式或脚本运行模式,因为在该环境中的主线程只有在退出 Python IDLE 时才终止。

示例 12-3 线程对象的 daemon 属性。




```
import threading
import time

class mythread(threading.Thread):        #继承 Thread 类,创建自定义线程类
    def __init__(self,num,threadname):
        threading.Thread.__init__(self,name=threadname)
        self.num=num
    def run(self):                       #重写 run()方法
        time.sleep(self.num)
        print(self.num)

t1=mythread(1,'t1')                      #创建自定义线程类对象,daemon 默认为 False
t2=mythread(5,'t2')
t2.daemon=True                           #设置线程对象 t2 的 daemon 属性为 True
print(t1.daemon)
print(t2.daemon)
t1.start()                               #启动线程
t2.start()
```

把上面的代码存储为 ThreadDaemon.py 文件，在 IDLE 环境中运行结果如图 12-3 所示，在命令提示符环境中运行结果如图 12-4 所示。可以看到，在命令提示符环境中执行该程序时，线程 t2 没有执行结束就跟随主线程一同结束了，因此并没有输出数字 5。

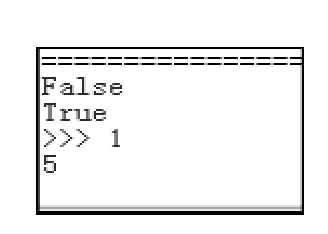


图 12-3　在 IDLE 环境中运行

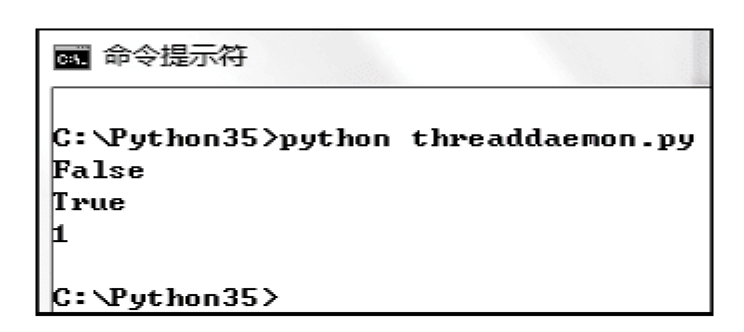


图 12-4　在命令提示符环境中运行

12.1.3　线程调度

在实际开发时，不仅要合理控制线程数量，还需要在多个线程之间进行有效地同步和调度才能发挥最大功效。

引入线程技术的主要目的之一是为了充分利用硬件资源，尤其是提高 CPU 的利用率，提高系统的任务处理速度和吞吐量，让各个部件都处于高速运转和忙碌状态。把任务拆分成能够互相协作的多个线程同时运行，那么属于同一个任务的多个线程之间必然会有交互和同步以便能够互相协作地完成任务。

在多核、多处理器平台上，在任意时刻每个核可以运行一个线程，多个线程同时运行并相互协作，从而达到高速处理任务的目的。然而，即使是高端服务器或工作站甚至集群系统，处理器和核的数量总是有限的，如果线程的数量多于核的数量，就必然需要进行调度来决定某个时刻哪些线程可以使用这些有限的资源。在调度时，处理器为每个线程分

配一个很短的时间片,所有线程根据具体的调度算法轮流获得该时间片。当时间片用完以后,即使该线程还没有执行完也要退出处理器等待下次调度,同时由操作系统按照优先级再选择一个线程进入 CPU 运行。在生命周期内,一个线程可能需要被调度并执行很多次才会结束,图 12-5 中椭圆内的上下文开关次数反映了线程被调度的次数。



图 12-5　WPS 进程中某个线程的属性

由于处理器中寄存器的数量有限,而不同的线程很可能需要使用到相同的一组寄存器来保存中间计算结果或运行状态。因此,在调度线程时必须要做好上下文的保存和恢复工作,以保证该线程下次被调度进处理器后能够继续上次的工作,而不是从头重来。虽然一般来说这些工作并不需要 Python 程序员操心,但是如果线程太多的话,线程调度带来的开销可能会比线程实际执行的开销还大,这样使用多线程就失去本来的意义了。

通过前面的图 12-3 可以发现,操作系统中会同时存在大量的线程,然而这并不意味着所有线程都是可调度的。虽然系统中同时会存在大量的线程,但是由于优先级或主动阻塞等原因,真正处于可调度状态的线程数量并不是非常多,系统也不会给暂时没事可做的线程分配任何 CPU 时间。例如,Python 标准库 time 中的 sleep( ) 函数,它的功能是暂停(或者说阻塞当前线程)指定时间(单位是秒)。从多线程编程和调度算法上来讲,该函数的真正功能是告诉操作系统在一定的时间内不要再调度自己了。

12.1.4　Lock/RLock 对象

Lock 是比较低级的同步原语,当被锁定以后不属于特定的线程。一个锁有两种状态:locked 和 unlocked,刚创建的 Lock 对象处于 unlocked 状态。如果锁处于 unlocked 状态,acquire( ) 方法将其修改为 locked 并立即返回;如果锁已处于 locked 状态,则阻塞当前线程并等待其他线程释放锁,然后将其修改为 locked 并立即返回。release( ) 方法用来将锁的状态由 locked 修改为 unlocked 并立即返回,如果锁状态本来已经是 unlocked,调用该



方法将会抛出异常。

可重入锁 RLock 对象也是一种常用的线程同步原语，可被同一个线程 acquire( ) 多次。当处于 locked 状态时，某线程拥有该锁；当处于 unlocked 状态时，该锁不属于任何线程。RLock 对象的 acquire( )/release( ) 调用对可以嵌套，仅当最后一个或者最外层的 release( ) 执行结束后，锁才会被设置为 unlocked 状态。

示例 12-4　使用 Lock/RLock 对象实现线程同步。

```
import threading
import time

#自定义线程类
class mythread(threading.Thread):
    def __init__(self):
        threading.Thread.__init__(self)
    #重写 run()方法
    def run(self):
        global x
        #获取锁,如果成功则进入临界区
        lock.acquire()
        x=x+3
        print(x)
        #退出临界区,释放锁
        lock.release()

lock=threading.RLock()
#也可以使用 Lock类实现加锁和线程同步
#lock=threading.Lock()

#存放多个线程的列表
tl=[]
for i in range(10):
    #创建线程并添加到列表
    t=mythread()
    tl.append(t)

#多个线程互斥访问的变量
x=0
#启动列表中的所有线程
for i in tl:
    i.start()
```

保存并运行上面的程序，依次输出 3、6、9、12、15、18、21、24、27、30 这几个数字。把 lock. acquire( ) 和 lock. release( ) 这两行注释或删除之后再运行，会发现多个线程之间没有任何“默契”，输出结果变得杂乱无章，每次运行可能会得到不同的结果。

需要注意的是,多线程同步时如果需要获得多个锁才能进入临界区的话,可能会发生死锁,在多线程编程时一定要注意并认真检查和避免这种情况。例如,下面的代码就有可能发生死锁,类似于“哲学家就餐问题”。

```
import threading
import time

class mythread1(threading.Thread):
    def __init__(self):
        threading.Thread.__init__(self)
    def run(self):
        lock1.acquire()                    #获取一个锁
        lock2.acquire()                    #获取另一个锁
        #实际功能代码(略)
        lock2.release()
        lock1.release()

class mythread2(threading.Thread):
    def __init__(self):
        threading.Thread.__init__(self)
    def run(self):
        lock2.acquire()
        lock1.acquire()
        #实际功能代码(略)
        lock1.release()
        lock2.release()

lock1 = threading.RLock()
lock2 = threading.RLock()
t1 = mythread1()
t2 = mythread2()
t1.start()
t2.start()
```

12.1.5 Condition 对象

使用 Condition 对象可以在某些事件触发后才处理数据或执行特定的功能代码,可以用于不同线程之间的通信或通知,以实现更高级别的同步。Condition 对象除了具有 acquire( )和 release( )方法之外,还有 wait( )、wait_for( )、notify( )、notify_all( )等方法。其中:

(1) wait(timeout = None)方法释放锁,并阻塞当前线程直到超时或其他线程针对同一个 condition 对象调用 notify( )/notify_all( )方法,被唤醒的线程会重新尝试获取锁并在成功获取锁之后结束 wait( )方法,然后继续执行。

（2）wait_for(predicate, timeout = None)方法阻塞当前线程直到超时或条件得到满足。

（3）notify(n = 1)唤醒等待该 condition 对象的一个或多个线程，该方法不负责释放锁。

（4）notify_all()方法会唤醒等待该 condition 对象的所有线程。

下面通过经典的生产者/消费者问题来演示 Condition 对象的用法，程序中生产者线程和消费者线程共享一个列表，生产者在列表尾部追加元素，消费者从列表首部获取并删除元素。如果列表长度到了 20 表示已满，生产者等待，如果列表已空则消费者等待。

示例 12-5　使用 Condition 对象实现线程同步。

```
import threading
from random import randint
from time import sleep

#自定义生产者线程类
class Producer(threading.Thread):
    def __init__(self,threadname):
        threading.Thread.__init__(self,name=threadname)
    def run(self):
        global x
        while True:
            #获取锁
            con.acquire()
            #假设共享列表中最多能容纳 20 个元素
            if len(x) ==20:
                #如果共享列表已满,生产者等待
                con.wait()
                print('Producer is waiting…')
            else:
                print('Producer:',end=' ')
                #产生新元素,添加至共享列表
                x.append(randint(1,1000))
                print(x)
                sleep(1)
                #唤醒等待条件的线程
                con.notify()
            #释放锁
            con.release()

#自定义消费者线程类
class Consumer(threading.Thread):
    def __init__(self,threadname):
        threading.Thread.__init__(self,name=threadname)
```

```
    def run(self):
        global x
        while True:
            #获取锁
            con.acquire()
            if not x:
                #等待
                con.wait()
                print('Consumer is waiting…')
            else:
                print(x.pop(0))
                print(x)
                sleep(2)
                con.notify()
            con.release()

#创建 Condition 对象以及生产者线程和消费者线程
con=threading.Condition()
x=[]
p=Producer('Producer')
c=Consumer('Consumer')
p.start()
c.start()
p.join()
c.join()
```

该程序运行结果如图 12-6 所示。

```
>>>
====================== RESTART: C:\Python 3.5\demo.py ========
>>> Producer: [842]
Producer: [842, 755]
Producer: [842, 755, 713]
Producer: [842, 755, 713, 73]
Producer: [842, 755, 713, 73, 166]
842
[755, 713, 73, 166]
Producer: [755, 713, 73, 166, 902]
Producer: [755, 713, 73, 166, 902, 921]
755
[713, 73, 166, 902, 921]
Producer: [713, 73, 166, 902, 921, 448]
713
[73, 166, 902, 921, 448]
Producer: [73, 166, 902, 921, 448, 53]
Producer: [73, 166, 902, 921, 448, 53, 865]
73
[166, 902, 921, 448, 53, 865]
```

图 12-6 使用 Condition 实现线程同步

12.1.6 Queue 对象

queue 模块中提供的 Queue 类实现多线程编程所需要的锁原语，是线程安全的，不需要额外的同步机制，尤其适合需要在多个线程之间进行信息交换的场合。Queue 类对象



的 get()和 put()方法都支持一个超时参数 timeout,调用该方法时如果超时会抛出异常。

示例 12-6　使用 Queue 对象实现多线程同步,模拟生产者/消费者问题。

```
import threading
import time
import queue

#自定义生产者线程类
class Producer(threading.Thread):
    def __init__(self,threadname):
        threading.Thread.__init__(self,name=threadname)
    def run(self):
        global myqueue
        #在队列尾部追加元素
        time.sleep(1)
        try:
            myqueue.put(self.getName(),timeout=1)
            print(self.getName(),' put ',self.getName(),' to queue.')
        except:
            pass

class Consumer(threading.Thread):
    def __init__(self,threadname):
        threading.Thread.__init__(self,name=threadname)
    def run(self):
        global myqueue
        #在队列首部获取元素
        time.sleep(0.1)
        try:
            print(self.getName(),' get ',myqueue.get(timeout=1.1),' from queue.')
        except:
            pass

myqueue=queue.Queue(5)

#创建生产者线程和消费者线程
plist=[]
clist=[]
for i in range(10):
    p=Producer('Producer'+str(i))
    plist.append(p)
    c=Consumer('Consumer'+str(i))
    clist.append(c)
```

```
#依次启动生产者线程和消费者线程
for p,c in zip(plist,clist):
    p.start()
    c.start()
```

示例 12-7 使用 Queue 类实现多线程复制文件，利用该类提供的机制来保证多个线程的同步。

```
import os
import sys
import argparse
from queue import Queue
from threading import Thread
from shutil import copyfile

def copyFile(src,dst,num):
    '''使用 num 个线程复制 src 目录下的文件到 dst 目录中'''

    #源文件夹必须存在
    assert os.path.isdir(src),src+' must be an existing directory.'
    #如果目标文件夹不存在，创建一个
    if not os.path.isdir(dst):
        os.makedirs(dst)
    #最多容纳 10 个元素的队列
    q=Queue(10)

    def add(src):
        for f in os.listdir(src):
            f=os.path.join(src,f)
            if os.path.isfile(f):
                #往队列中放数据，满了会自动等待
                q.put(f)
            elif os.path.isdir(f):
                q.put(f)
                add(f)

    #创建并启动往队列中存放元素的线程
    t_add=Thread(target=add,args=(src,))
    t_add.start()

    def copy():
        while True:
            srcItem=q.get()
            if srcItem==None:
                break
```

```
            #替换字符串,生成目标路径
            dstItem=srcItem.replace(src,dst)
            print(srcItem,'==>',dstItem)
            if os.path.isfile(srcItem):
                dstDir=os.path.split(dstItem)[0]
                if not os.path.isdir(dstDir):
                    try:
                        os.makedirs(dstDir)
                    except FileExistsError as e:
                        pass
                copyfile(srcItem,dstItem)
            elif os.path.isdir(srcItem):
                try:
                    os.makedirs(dstItem)
                except FileExistsError as e:
                    pass
            #发送完成信号
            q.task_done()

    #创建指定数量的线程来复制文件
    for i in range(num):
        t=Thread(target=copy)
        t.start()

    #等待所有任务完成
    q.join()

    #往队列中插入 None 空值,让线程结束
    for i in range(num):
        q.put(None)

if __name__=='__main__':

    #解析命令行参数
    parser=argparse.ArgumentParser(description='copy files from src to dst')
    parser.add_argument('-s','--src')#default='C:\\python35')
    parser.add_argument('-d','--dst')#default='D:\\test')
    parser.add_argument('-n','--num',default='5')
    args=parser.parse_args()

    if args.src!=None and args.dst!=None:
        copyFile(args.src,args.dst,int(args.num))
    else:
        print('Please use the following command to see how to use:')
```

```
        print('    '+sys.argv[0]+' -h')
```

把上面的代码保存为 multiThread_copyFile. py,用法如下:

```
C:\Python35>python multiThread_copyFile.py -h
usage: multiThread_copyFile.py [-h] [-s SRC] [-d DST] [-n NUM]

copy files from src to dst

optional arguments:
  -h, --help            show this help message and exit
  -s SRC, --src SRC
  -d DST, --dst DST
  -n NUM, --num NUM
```

12.1.7 Event 对象

Event 对象是一种简单的线程通信技术,一个线程设置 Event 对象,另一个线程等待 Event 对象。Event 对象的 set()方法可以设置 Event 对象内部的信号标志为真;clear()方法可以清除 Event 对象内部的信号标志,将其设置为假;isSet()方法用来判断其内部信号标志的状态;wait()方法在其内部信号状态为真时会立刻执行并返回,若 Event 对象的内部信号标志为假,wait()方法就一直等待至超时或者内部信号状态为真。

示例 12-8 使用 Event 对象实现线程同步。

```
import threading

#自定义线程类
class mythread(threading.Thread):
    def __init__(self,threadname):
        threading.Thread.__init__(self,name=threadname)

    def run(self):
        global myevent
        #根据 Event 对象是否已设置做出不同的响应
        if myevent.isSet():
            #清除标志
            myevent.clear()
            #等待
            myevent.wait()
            print(self.getName()+' set')
        else:
            print(self.getName()+' not set')
            #设置标志
            myevent.set()
```

```
myevent = threading.Event()
#设置标志
myevent.set()

for i in range(10):
    t = mythread(str(i))
    t.start()
```

将上面的代码保存为 demo.py 文件并多次运行，会发现每次运行结果略有不同，图 12-7 是其中两次的运行结果。

命令提示符	命令提示符
C:\Python36>python demo.py	C:\Python36>python demo.py
1 not set	1 not set
0 set	0 set
3 not set	3 not set
2 set	2 set
5 not set	5 not set
4 set	6 not set
7 not set	4 set
6 set	7 set
9 not set	9 not set
8 set	8 set

图 12-7　使用 Event 对象实现线程同步

12.1.8　Semaphore 与 BoundedSemaphore

Semaphore 对象维护着一个内部计数器，调用 acquire( ) 方法时该计数器减 1，调用 release( ) 方法时该计数器加 1，适用于需要控制特定资源的并发访问线程数量的场合。调用 acquire( ) 方式时，如果计数器已经为 0 则阻塞当前线程直到有其他线程调用了 release( ) 方法，所以计数器的值永远不会小于 0。Semaphore 对象可以调用任意次 release ( ) 方法（如果真的出现这种情况，很可能是有 bug），而 BoundedSemaphore 对象可以保证计数器的值不超过特定的值。与 Lock/RLock、Condition 对象一样，Semaphore 和 BoundedSemaphore 对象也支持上下文管理协议，支持 with 关键字。

示例 12-9　使用 BoundedSemaphore 对象限制特定资源的并发访问线程数量。

```
import threading
import time

def worker(value):
    with sema:
        print(value)
        time.sleep(8)

#同一时刻最多允许 2 个线程访问特定资源
sema = threading.BoundedSemaphore(2)
```

```
for i in range(10):
    t=threading.Thread(target=worker,args=(i,))
    t.start()
```

12.1.9 Barrier 对象

Barrier 对象常用来实现这样的线程同步，多个线程运行到某个时间点以后每个线程都需要等着其他线程都准备好以后再同时进行下一步工作。类似于赛马时需要先用栅栏拦住，每个试图穿过栅栏的选手都需要明确说明自己准备好了，当所有选手都表示准备好以后，栅栏打开，所有选手同时冲出栅栏。

下面的代码创建了一个允许 3 个线程互相等待的 Barrier 对象，每个线程做完一些准备工作后调用 Barrier 对象的 wait( )方法等待其他线程，当所有线程都调用了 wait( )方法之后，会调用指定的 action 对象，然后同时开始执行 wait( )之后的代码。

```
import threading
import random
import time

def worker(arg):
    #假设每个线程需要不同的时间来完成准备工作
    time.sleep(random.randint(1,20))
    #假设已知任何线程的准备工作最多需要 20s
    #每个线程调用 wait()时,返回值不一样
    r=b.wait(20)
    if r==0:
        print(arg)

def printOk():
    print('ok')

#允许 3 个线程等待
#如果线程调用 wait()时没有指定超时时间,默认为 20s
b=threading.Barrier(parties=3,action=printOk,timeout=20)

#创建并启动 3 个线程,线程数量必须与 Barrier 对象的 parties 一致
for i in range(3):
    t=threading.Thread(target=worker,args=(i,))
    t.start()
```

下面的代码模拟了一个类似的场景，服务器启动时需要一定的时间，在服务器做好准备工作之前不允许客户端发起连接请求。

```
import threading
import random
```



```
import time

b=threading.Barrier(2,timeout=5)

def server():
    #启动服务器,准备接收客户端连接,代码略
    b.wait()
    while True:
        #接收客户端连接,处理客户端请求,代码略

def client():
    #等待服务器启动
    b.wait()
    while True:
        #建立连接,和服务器进行通信

#分别创建并启动服务器线程和客户端线程
threading.Thread(target=server).start()
threading.Thread(target=client).start()
```

12.2　多进程编程

进程是正在执行中的应用程序。一个进程是正在执行中的一个程序使用资源的总和,包括虚拟地址空间、代码、数据、对象句柄、环境变量和执行单元等。一个应用程序同时打开并执行多次,就会创建多个进程。

Python 标准库 multiprocessing 用来实现进程的创建与管理以及进程间的同步与数据交换,用法与 threading 类似,是支持并行处理的重要模块。标准库 multiprocessing 同时支持本地并发与远程并发,有效避免了全局解释器锁(Global Interpreter Lock,GIL)问题,可以更有效地利用 CPU 资源,尤其适合多核或多 CPU 环境。

12.2.1　进程创建与管理

与使用 threading 中的 Thread 类创建线程对象类似,可以通过 multiprocessing 中的 Process 类来创建一个进程对象,然后通过调用进程对象的 start()方法来启动,通过调用 join()方法等待一个进程执行结束。

示例 12-10　进程创建与启动。

```
from multiprocessing import Process
import os

def f(name):
    print('module name:',__name__)
```

```
    print('parent process:',os.getppid())        #查看父进程的 ID
    print('process id:',os.getpid())             #查看当前进程的 ID
    print('hello',name)

if __name__=='__main__':
    p=Process(target=f,args=('bob',))           #创建进程
    p.start()                                   #启动进程
    p.join()                                    #等待进程运行结束
```

标准库 multiprocessing 中的很多功能都要求在命名空间__main__中运行,因此在本节代码中的"if __name__ = = '__main__':"最好都要保留,否则会引发异常而无法正常执行。另外,本节的很多程序需要在命令提示符环境中运行才能正常执行并得到预期结果。

标准库 os 中提供了一些查看进程属性的函数,示例 12-10 演示了几个。扩展库 psutil 中提供了更多的有关方法,附录 H 中介绍了查看和结束特定进程的用法。

12.2.2 进程同步技术

在需要协同工作完成大型任务时,多个进程间的同步非常重要。进程同步方法与线程同步方法类似,代码稍微改写一些即可,下面以 Lock 对象和 Event 对象为例简单演示其用法。

示例 12-11 使用 Lock 对象实现进程同步。

```
from multiprocessing import Process,Lock

def f(lock,i):
    with lock:                                  #Lock 对象支持上下文管理协议
        print('hello world',i)

if __name__=='__main__':
    lock=Lock()                                 #创建锁对象
    for num in range(10):
        Process(target=f,args=(lock,num)).start()
```

示例 12-12 使用 Event 对象实现进程同步。

```
from multiprocessing import Process,Event

def f(e,i):
    if e.is_set():
        e.wait()
        print('hello world',i)
        e.clear()
    else:
        e.set()
```




```
if __name__=='__main__':
    e=Event()
    for num in range(10):
        Process(target=f,args=(e,num)).start()
```

12.2.3　Pool 对象

除了支持与 threading 管理线程相似的接口之外，multiprocessing 还提供了 Pool 对象支持数据的并行操作。Pool 对象提供了大量的方法支持并行操作，常用的有：

（1）apply(func[, args[, kwds]])：调用函数 func，并传递参数 args 和 kwds，同时阻塞当前进程直至函数返回，函数 func 只会在进程池中的一个工作进程中运行。

（2）apply_async(func[, args[, kwds[, callback[, error_callback]]]])：apply()的变形，返回结果对象，可以通过结果对象的 get()方法获取其中的结果；参数 callback 和 error_callback 都是单参数函数，当结果对象可用时会自动调用 callback，该调用失败时会自动调用 error_callback。

（3）map(func, iterable[, chunksize])：内置函数 map()的并行版本，但只能接收一个可迭代对象作为参数，该方法会阻塞当前进程直至结果可用。该方法会把迭代对象 iterable 切分成多个块再作为独立的任务提交给进程池，块的大小可以通过参数 chunksize（默认值为 1）来设置。

（4）map_async(func, iterable[, chunksize[, callback[, error_callback]]])：与 map()方法类似，但返回结果对象，需要使用结果对象的 get()方法来获取其中的值。

（5）imap(func, iterable[, chunksize])：map()方法的惰性求值版本，返回迭代器对象。

（6）imap_unordered(func, iterable[, chunksize])：与 imap()方法类似，但不保证结果会按参数 iterable 中原来元素的先后顺序返回。

（7）starmap(func, iterable[, chunksize])：类似于 map()方法，但要求参数 iterable 中的元素为迭代对象并可解包为函数 func 的参数。

（8）starmap_async(func, iterable[, chunksize[, callback[, error_back]]])：方法 starmap()和 map_async()的组合，返回结果对象。

（9）close()：不允许再向进程池提交任务，当所有已提交任务完成后工作进程会退出。

（10）terminate()：立即结束工作进程，当线程池对象被回收时会自动调用该方法。

（11）join()：等待工作进程退出，在此之前必须先调用 close()或 terminate()。

示例 12-13　并发计算二维数组每行的平均值。

```
from multiprocessing import Pool
from statistics import mean

def f(x):
    return mean(x)
```

```
if __name__=='__main__':
    x=[list(range(10)),list(range(20,30)),
       list(range(50,60)),list(range(80,90))]
    with Pool(5) as p:                          #创建包含 5 个进程的进程池
        print(p.map(f,x))                       #并发运行
```

示例 12-14 并行计算列表中数值的平方值。

```
from multiprocessing import Pool

def f(x):
    return x*x

if __name__=='__main__':
    with Pool(5) as p:
        print(p.map(f,[1,2,3]))
```

示例 12-15 并行判断 100 000 000 以内的数字是否为素数,并统计素数的个数。

```
from multiprocessing import Pool

def isPrime(n):
    if n<2:
        return 0
    if n==2:
        return 1
    if not n&1:
        return 0
    for i in range(3,int(n**0.5)+1,2):
        if n%i==0:
            return 0
    return 1

if __name__=='__main__':
    with Pool(5) as p:
        print(sum(p.map(isPrime,range(100000000))))
```

示例 12-16 Pool 对象几种常用方法的用法。

```
from multiprocessing import Pool
import time

def f(x):
    return x*x

if __name__=='__main__':
    with Pool(processes=4) as pool:
```



```
    #返回结果对象,可以通过 get()方法获取其中的值
    result =pool.apply_async(f, (10,))
    print(result.get(timeout =1))
    #直接返回结果列表
    print(pool.map(f,range(10)))
    #返回迭代器对象
    it =pool.imap(f,range(10))
    print(next(it))
    print(next(it))
    print(it.next(timeout =1))
    #进入睡眠状态 10s
    result =pool.apply_async(time.sleep, (10,))
    #下面的代码会引发超时异常
    print(result.get(timeout =3))
```

示例 12-17　Pool 对象的方法在不同情况下引发异常的处理方法。

```
import multiprocessing
import time
import random
import sys

def calculate(func,args):
    #调用 mul 或 plus 函数计算结果并返回格式化后的字符串
    #括号里的 * 表示序列解包
    result =func(*args)
    return '{3} says {1}{2} ={0}'.format(func.__name__,args,result,
        multiprocessing.current_process().name)

def calculatestar(args):
    return calculate(*args)

def mul(a,b):
    time.sleep(0.5 *random.random())
    return a *b

def plus(a,b):
    time.sleep(0.5 *random.random())
    return a +b

def f(x):
    #当 x=5 时该函数会引发异常
    return 1.0/(x -5.0)

def test():
```

```
#创建包含 4 个进程的进程池
with multiprocessing.Pool(4) as pool:

    TASKS =[(mul,(i,7)) for i in range(10)] +\
           [(plus,(i,8)) for i in range(10)]

    print('Testing error handling:')
    try:
        print(pool.apply(f,(5,)))
    except ZeroDivisionError:
        print('\tGot ZeroDivisionError as expected from pool.apply()')
    else:
        raise AssertionError('expected ZeroDivisionError')

    try:
        print(pool.map(f,list(range(10))))
    except ZeroDivisionError:
        print('\tGot ZeroDivisionError as expected from pool.map()')
    else:
        raise AssertionError('expected ZeroDivisionError')

    try:
        print(list(pool.imap(f,list(range(10)))))
    except ZeroDivisionError:
        print('\tGot ZeroDivisionError as expected from list(pool.imap())')
    else:
        raise AssertionError('expected ZeroDivisionError')

    it =pool.imap(f,list(range(10)))
    for i in range(10):
        try:
            x =next(it)
        except ZeroDivisionError:
            if i ==5:
                pass
        except StopIteration:
            break
        else:
            if i ==5:
                raise AssertionError('expected ZeroDivisionError')

    assert i ==9
    print('\tGot ZeroDivisionError as expected from IMapIterator.next()')
    print()
```



```
    #测试超时是否正常
    print('Testing ApplyResult.get() with timeout:',end=' ')
    res =pool.apply_async(calculate,TASKS[0])
    while True:
        sys.stdout.flush()
        try:
            sys.stdout.write('\n\t%s' % res.get(0.02))
            break
        except multiprocessing.TimeoutError:
            sys.stdout.write('.')
    print()

    print('Testing IMapIterator.next() with timeout:',end=' ')
    it =pool.imap(calculatestar,TASKS)
    while True:
        sys.stdout.flush()
        try:
            sys.stdout.write('\n\t%s' % it.next(0.02))
        except StopIteration:
            break
        except multiprocessing.TimeoutError:
            sys.stdout.write('.')
    print()

if __name__=='__main__':
    multiprocessing.freeze_support()
    test()
```

12.2.4 Manager 对象

Manager 对象提供了不同进程间共享数据的方式，甚至可以在网络上不同机器上运行的进程间共享数据。Manager 对象控制一个拥有 list、dict、Lock、RLock、Semaphore、BoundedSemaphore、Condition、Event、Barrier、Queue、Value、Array、Namespace 等对象的服务端进程，并且允许其他进程通过代理来操作这些对象。

示例 12-18 使用 Manager 对象实现进程间数据交换。

```
from multiprocessing import Process,Manager

def f(d,l,t):
    d['name'] ='Dong Fuguo'
    d['age'] =38
    d['sex'] ='Male'
    d['address'] ='Yantai'
    l.reverse()
```

```
    t.value=3
if __name__=='__main__':
    with Manager() as manager:
        d=manager.dict()
        l=manager.list(range(10))
        t=manager.Value('i',0)
        p=Process(target=f,args=(d,l,t))
        p.start()
        p.join()
        for item in d.items():
            print(item)
        print(l)
        print(t.value)
```

示例 12-19 使用 Manager 对象实现不同机器上的进程跨网络共享数据。

(1) 编写程序文件 multiprocessing_server.py,启动服务器进程,创建可共享的队列对象。

```
from multiprocessing.managers import BaseManager
from queue import Queue

q=Queue()
class QueueManager(BaseManager):
    pass
QueueManager.register('get_queue',callable=lambda:q)

m=QueueManager(address=('',30030),authkey=b'dongfuguo')
s=m.get_server()
s.serve_forever()
```

(2) 编写程序文件 multiprocessing_client1.py,连接服务器进程,并往共享的队列中存入一些数据。

```
from multiprocessing.managers import BaseManager

class QueueManager(BaseManager):
    pass
QueueManager.register('get_queue')
#假设服务器的 IP 地址为 10.2.1.2
m=QueueManager(address=('10.2.1.2',30030),authkey=b'dongfuguo')
m.connect()
q=m.get_queue()
for i in range(3):
    q.put(i)
```

(3) 编写程序文件 multiprocessing_client2.py,连接服务器进程,从共享的队列对象中



读取数据并输出显示。

```
from multiprocessing.managers import BaseManager

class QueueManager(BaseManager):
    pass
QueueManager.register('get_queue')
m=QueueManager(address=('10.2.1.2',30030),authkey=b'dongfuguo')
m.connect()
q=m.get_queue()
for i in range(3):
    print(q.get())
```

示例 12-20　创建和使用自定义 Manager 对象与 Proxy 对象。

```
from multiprocessing import freeze_support
from multiprocessing.managers import BaseManager,BaseProxy
import operator

#普通类
class Foo:
    def f(self):
        print('you called Foo.f()')
    def g(self):
        print('you called Foo.g()')
    def _h(self):
        print('you called Foo._h()')

#生成器
def baz():
    for i in range(10):
        yield i * i

#生成器对象的代理类
class GeneratorProxy(BaseProxy):
    _exposed_=['__next__']
    def __iter__(self):
        return self
    def __next__(self):
        return self._callmethod('__next__')

#返回 operator 模块的函数
def get_operator_module():
    return operator
```

```
class MyManager(BaseManager):
    pass

#注册 Foo 类,默认的公开成员 f()和 g()可以通过代理访问
MyManager.register('Foo1',Foo)

#注册 Foo 类,明确指定成员 g()和_h()可以通过代理来访问
MyManager.register('Foo2',Foo,exposed=('g','_h'))

#注册生成器函数 baz,指定代理类型为 GeneratorProxy
MyManager.register('baz',baz,proxytype=GeneratorProxy)

#注册函数 get_operator_module(),使其可以通过代理进行访问
MyManager.register('operator',get_operator_module)

def test():
    manager=MyManager()
    manager.start()

    print('-'*20)
    #创建对象
    f1=manager.Foo1()
    #调用对象成员
    f1.f()
    f1.g()
    #确认对象拥有哪些可访问的成员
    assert not hasattr(f1,'_h')
    assert sorted(f1._exposed_)==sorted(['f','g'])

    print('-'*20)
    f2=manager.Foo2()
    f2.g()
    f2._h()
    assert not hasattr(f2,'f')
    assert sorted(f2._exposed_)==sorted(['g','_h'])

    print('-'*20)
    it=manager.baz()
    for i in it:
        print('<%d>'%i,end=' ')
    print()

    print('-'*20)
    op=manager.operator()
```




```
    print('op.add(23,45) =',op.add(23,45))
    print('op.pow(2,94) =',op.pow(2,94))
    print('op._exposed_ =',op._exposed_)

if __name__=='__main__':
    #支持使用 py2exe、Pyinstaller 和 cx_Freeze 打包为 Windows 可执行程序
    freeze_support()
    test()
```

12.2.5　Listener 与 Client 对象

这两个对象是 multiprocessing. connection 模块提供的对象,可以在不同机器上的进程之间通过网络直接传输整数、实数、字符串、列表、元组、数组等各种类型的信息。

示例 12-21　使用 Listener 于 Client 对象在不同机器之间传递信息,用来验证服务端是否存活。

服务端或监听端程序代码如下：

```
from multiprocessing.connection import Listener
from time import sleep

with Listener(('',6060),authkey=b'dongfuguo') as listener:
    with listener.accept() as conn:
        print('connection accepted from',listener.last_accepted)
        i =0
        while True:
            conn.send(('Server is alive',i))
            i +=1
            sleep(3)
```

客户端代码如下：

```
from multiprocessing.connection import Client

with Client(('10.2.1.2',6060),authkey=b'dongfuguo') as conn:
    while True:
        print(conn.recv())
```

12.2.6　进程间数据交换与共享

除了使用前几节介绍的 Manager 对象、Listener 与 Client 对象在不同进程之间进行数据交换与共享,Queue 对象和 Pipe 也常用来完成类似任务。

示例 12-22　使用 Queue 对象在进程间交换数据,一个进程把数据放入 Queue 对象,另一个进程从 Queue 对象中获取数据。使用 Queue 对象在进程间交换信息时,必须要保证放入队列中的所有数据都被取走,否则可能会导致死锁。

```
import multiprocessing as mp

def foo(q):
    q.put('hello world!')                    #把数据放入队列

if __name__=='__main__':
    mp.set_start_method('spawn')             #Windows 系统创建子进程的默认方式
    q=mp.Queue()
    p=mp.Process(target=foo,args=(q,))       #创建进程,把 Queue 对象作为参数传递
    p.start()
    print(q.get())                           #从队列中获取数据
    p.join()
```

下面的代码演示了更加复杂的情形,当前进程负责向队列中提交任务,子进程负责进行相应的计算,并通过另一个队列把计算结果返回给当前进程。

```
import time
import random
from multiprocessing import Process,Queue,current_process,freeze_support

def worker(task_queue,output):
    #持续执行 task_queue.get(),直到取到'STOP'
    for func,args in iter(task_queue.get,'STOP'):
        #调用相应的函数,并将计算结果放入队列
        result=calculate(func,args)
        output.put(result)

def calculate(func,args):
    result=func(*args)
    return '%s says that %s%s=%s' % \
        (current_process().name,func.__name__,args,result)

def mul(a,b):
    time.sleep(0.5 * random.random())
    return a*b

def plus(a,b):
    time.sleep(0.5 * random.random())
    return a+b

def test():
    NUMBER_OF_PROCESSES=4
    TASKS1=[(mul,(i,7)) for i in range(10)]
    TASKS2=[(plus,(i,8)) for i in range(10)]
```




```
    #创建 Queue 对象
    task_queue=Queue()
    done_queue=Queue()

    #创建并启动进程
    for i in range(NUMBER_OF_PROCESSES):
        Process(target=worker,args=(task_queue,done_queue)).start()

    #提交任务
    for task in TASKS1:
        task_queue.put(task)

    #输出计算结果
    print('Unordered results:')
    for i in range(len(TASKS1)):
        print('\t',done_queue.get())

    #提交更多任务
    for task in TASKS2:
        task_queue.put(task)

    #输出更多结果
    for i in range(len(TASKS2)):
        print('\t',done_queue.get())

    #发送停止信号
    for i in range(NUMBER_OF_PROCESSES):
        task_queue.put('STOP')

if __name__=='__main__':
    freeze_support()
    test()
```

另外，也可以使用上下文对象 context 的 Queue 对象实现不同进程间的数据交换。

```
import multiprocessing as mp

def foo(q):
    q.put('hello world')

if __name__=='__main__':
    ctx=mp.get_context('spawn')
    q=ctx.Queue()
    p=ctx.Process(target=foo,args=(q,))
    p.start()
```

```
    print(q.get())
    p.join()
```

示例 12-23 使用管道实现进程间的数据交换。管道有两个端,一个接收端和一个发送端,相当于在两个进程之间建立了一个用于传输数据的通道。

```
from multiprocessing import Process,Pipe

def f(conn):
    conn.send('hello world')                #向管道中发送数据
    conn.close()                            #关闭管道

if __name__=='__main__':
    parent_conn,child_conn=Pipe()           #创建管道对象
    p=Process(target=f,args=(child_conn,))  #将管道的一方作为参数传递给子进程
    p.start()
    p.join()
    print(parent_conn.recv())               #通过管道的另一方获取数据
    parent_conn.close()
```

示例 12-24 使用共享内存实现进程间数据传递,比较适合大量数据的场合。

```
from multiprocessing import Process,Value,Array

def f(n,a):
    n.value=3.1415927
    for i in range(len(a)):
        a[i]=a[i] * a[i]

if __name__=='__main__':
    num=Value('d',0.0)                      #实型
    arr=Array('i',range(10))                #整型数组
    p=Process(target=f,args=(num,arr))      #创建进程对象
    p.start()
    p.join()
    print(num.value)
    print(arr[:])
```

12.2.7 标准库 subprocess

标准库 subprocess 允许创建子进程,连接子进程的输入输出管道,并获得子进程的返回码,也是常用的并发执行技术之一。该标准库提供了 run()、call()和 Popen()3 种不同的函数用来创建子进程,其中 run()函数会阻塞当前进程,子进程结束后返回包含返回码和其他信息的 CompletedProcess 对象;call()函数也会阻塞当前进程,子进程结束后直接得到返回码;Popen()函数创建子进程时不阻塞当前进程,直接返回得到 Popen 对象,通过



该对象可以对子进程进行更多的操作和控制。例如，Popen 对象的 kill( ) 和 terminate( ) 方法可以用来结束该进程，send_signal( ) 可以给子进程发送指定信号，wait( ) 方法用来等待子进程运行结束，pid 用来表示子进程的 ID 号，等等。

```
>>>p=subprocess.Popen('c:\\windows\\notepad.exe')    #创建并运行子进程
>>>p.pid                                               #进程 ID
2744
>>>p.kill()                                            #结束进程
```

12.3　协　　程

协程的概念有两种含义：①用来定义协程的函数，此时也可称为协程函数；②调用协程函数得到的协程对象，表示一个最终会完成的计算或者 I/O 操作。

协程的引入使得编写单线程并发代码成为可能，事件循环在单个线程中运行并在同一个线程中执行所有的回调函数和任务，当事件循环中正在运行一个任务时，该线程中不会再同时运行其他任务，一个事件循环在某个时刻只运行一个任务。但是如果该任务执行 yield from 语句等待某个 Future 对象的完成，则当前任务被挂起，事件循环执行下一个任务。当然，不同线程中的事件循环可以并行执行多个任务。

在语法形式上，协程可以通过 async def 语句或生成器来实现，如果不需要考虑和旧版本 Python 兼容的话，应优先考虑前者；基于生成器的协程函数需要使用@ asyncio.coroutine 进行修饰，并且使用 yield from 而不是 yield 语句。

Future 类代表可调用对象的异步执行，Task 类是 Future 的子类，用来调度协程，负责在事件循环中执行协程对象，如果在协程中使用 yield from 语句从一个 Future 对象中返回值的话，Task 对象会挂起协程的执行并且等待 Future 对象的完成，当 Future 对象完成后，协程会重新启动并得到 Future 对象的结果或异常。

与普通函数不同，调用一个协程函数并不会立刻启动代码的执行，返回的协程对象在被调度之前不会做什么事情。启动协程对象的执行有两种方法：①在一个正在运行的协程中使用 await 或者 yield from 语句等待协程对象的返回结果；②使用 ensure_future( ) 函数或者 AbstractEventLoop. create_task( ) 方法创建任务(Task 对象) 并调度协程的执行。

示例 12-25　在单线程中使用事件循环同时计算多个整数的阶乘。

```
import asyncio

async def factorial(name,number):
    f=1
    for i in range(2,number+1):
        print("Task %s: Compute factorial(%s)..." % (name,i))
        await asyncio.sleep(0.5)
        f*=i
    print("Task %s: factorial(%s) =%s" % (name,number,f))
```

```
loop=asyncio.get_event_loop()
tasks=[asyncio.ensure_future(factorial("A",14)),
       asyncio.ensure_future(factorial("B",13)),
       asyncio.ensure_future(factorial("C",16))]
loop.run_until_complete(asyncio.gather(*tasks))
loop.close()
```

在上面的代码中，asyncio.get_event_loop()函数用来返回当前上下文中实现AbstractEventLoop接口的事件循环对象。asyncio.gather()函数用来返回一个从给定的协程对象或Future对象得到的聚集结果，要求所有的Future对象共享同一个事件循环，如果所有任务都顺利完成，该函数返回结果列表。

示例 12-26 显示当前日期时间。

```
import asyncio.subprocess
import sys

@asyncio.coroutine
def get_date():
    code='import datetime; print(datetime.datetime.now())'

    #创建子进程，并把标准输出重定向到管道
    create=asyncio.create_subprocess_exec(sys.executable,'-c',code,
                                          stdout=asyncio.subprocess.PIPE)
    proc=yield from create

    #读取一行输出
    data=yield from proc.stdout.readline()
    line=data.decode('ascii').rstrip()

    #等待子进程退出
    yield from proc.wait()
    return line

if sys.platform=="win32":
    loop=asyncio.ProactorEventLoop()
    asyncio.set_event_loop(loop)
else:
    loop=asyncio.get_event_loop()

date=loop.run_until_complete(get_date())
print("Current date: %s" %date)
loop.close()
```

示例 12-27 使用协程计算阶乘。




```
import asyncio
import operator
import functools

@asyncio.coroutine
def slow_operation(future,n):
    yield from asyncio.sleep(1)
    result =functools.reduce(operator.mul,range(1,n +1))
    #设置计算结果
    future.set_result(result)

loop =asyncio.get_event_loop()
future =asyncio.Future()
#创建并启动任务,计算 50 的阶乘
asyncio.ensure_future(slow_operation(future,50))
loop.run_until_complete(future)
#输出计算结果
print(future.result())
loop.close()
```

示例 12-28　在事件循环中执行函数。

```
import asyncio

def hello_world(loop):
    print('Hello World')
    #结束事件循环
    loop.stop()

loop =asyncio.get_event_loop()

#在指定的事件循环中执行函数
loop.call_soon(hello_world,loop)

#一直运行事件循环,阻塞当前线程,直到调用 loop.stop()
loop.run_forever()
loop.close()
```

12.4　concurrent.futures 模块提供的并发执行功能

concurrent.futures 模块提供了异步执行的高级接口,可以通过 ThreadPoolExecutor 实现线程的异步执行,也可以通过 ProcessPoolExecutor 实现进程的异步执行,两者都继承自抽象类 Executor,提供了相同的接口。

(1) submit(fn, *args, **kwargs)用来调度可调用对象 fn 并为其传递参数 args 和

kwargs,返回一个 Future 对象。

(2) map(func, *iterables, timeout = None, chunksize = 1)是与内置函数 map(func, *iterables)等价的异步执行方法,多个 func 的调用可以并发执行。

(3) shutdown(wait = True)方法通知 Executor 对象执行完当前 Future 对象之后释放所有资源,如果参数 wait 为 True,则 shutdown()方法等待执行结束并释放有关资源之后再返回,否则立即返回。

示例 12-29 使用 ThreadPoolExecutor 把 C:\test 中的所有文件批量复制 D:\test 文件夹,假设目标文件夹 D:\test 已存在。

```
from concurrent.futures import ThreadPoolExecutor
from shutil import copy
from os import listdir
from os.path import isfile,join

with ThreadPoolExecutor(max_workers=4) as e:
    for f in (fn for fn in listdir('C:\\test')):
        src=join('C:\\test',f)
        if isfile(src):
            dst=join('D:\\test',f)
            e.submit(copy,src,dst)
```

示例 12-30 使用 ProcessPoolExecutor 批量快速判断素数。

```
from concurrent.futures import ProcessPoolExecutor

PRIMES=[1099726899285419,112582705942171,
        112272535095293,115280095190773,
        115797848077099,9000099011]

def isPrime(n):
    if n%2==0:
        return False

    for i in range(3,int(n**0.5)+1,2):
        if n%i==0:
            return False
    return True

def main():
    with ProcessPoolExecutor() as executor:
        for number,prime in zip(PRIMES,executor.map(isPrime,PRIMES)):
            print('%d is prime: %s' % (number,prime))

if __name__=='__main__':
```




```
main()
```

12.5　pySpark 并行计算与分布式计算框架

Spark 是一个**开源的、通用的并行计算与分布式计算框架**，其活跃度在 Apache 基金会所有开源项目中排第三位，最大特点是**基于内存计算**，适合迭代计算，兼容多种应用场景，同时还兼容 Hadoop 生态系统中的组件，并且具有非常强的容错性。Spark 的设计目的是全栈式解决批处理、结构化数据查询、流计算、图计算和机器学习等业务和应用，适用于需要多次操作特定数据集的应用场合。需要反复操作的次数越多，所需读取的数据量越大，效率提升越大。

Spark 集成了 Spark SQL（分布式 SQL 查询引擎，提供了一个 DataFrame 编程抽象）、Spark Streaming（把流式计算分解成一系列短小的批处理计算，并且提供高可靠和吞吐量服务）、MLlib（提供机器学习服务）、GraphX（提供图计算服务）、SparkR（R on Spark）等子框架，为不同应用领域的从业者提供了全新的大数据处理方式，越来越便捷、轻松。

为了适应迭代计算，Spark 把经常被重用的数据缓存到内存中以提高数据读取和操作速度，比 Hadoop 快近百倍，并且支持 Java、Scala、Python、R 等多种语言。除 map 和 reduce 之外，Spark 还支持 filter、foreach、reduceByKey、aggregate 以及 SQL 查询、流式查询等。

随着普通家用计算机（手机也早已进入多核时代，但如何在手机上搭建 Spark 环境不在本书讨论范围之内）进入多处理器和多核时代，完全可以在自己家的计算机上搭建 Spark 环境。当然，如果数据量大到一定程度的话，还是要在集群或云平台上部署的 Spark 环境中进行处理和计算。进行 Spark 应用开发时一般是先在本地进行开发和测试，通过测试后再提交到集群执行。下面我们以 Windows 7 平台为例介绍 Spark 环境的搭建和简单使用。首先安装 JDK 并配置环境变量 path，下载安装 Scala 语言包并配置系统环境变量 path，下载安装 Spark 并配置系统环境变量 HADOOP_HOME 和 SPARK_HOME 的值为 Spark 安装目录，使用 pip 工具安装扩展库 py4j，到网址 http://public-repo-1.hortonworks.com/hdp-win-alpha/winutils.exe 下载 winutils.exe 放到 Spark 安装目录的 bin 文件夹中，最后进入命令提示符环境并切换到 Spark 安装目录的 bin 子文件夹，执行命令 pyspark.cmd，进入 Python 开发环境，如图 12-8 所示。可以看到，Spark 不仅可以使用 pyspark 库，还可以使用 Python 标准库和已安装的扩展库。

扩展库 pyspark 提供了 SparkContext（Spark 功能的主要入口，一个 SparkContext 表示与一个 Spark 集群的连接，可用来创建 RDD 或在该集群上广播变量）、RDD（Spark 中的基本抽象，弹性分布式数据集 Resilient Distributed Dataset）、Broadcast（可以跨任务重用的广播变量）、Accumulator（共享变量，任务只能为其增加值）、SparkConf（用来配置 Spark）、SparkFiles（访问任务的文件）、StorageLevel（更细粒度的缓冲永久级别）等可以公开访问的类，并且提供了 pyspark.sql、pyspark.streaming 与 pyspark.mllib 等模块与包。

另外，在 Spark 的 bin 文件夹中还提供了 spark-submit.cmd 文件，这个文件是用来执行 Python 程序的，使用任意 Python 开发环境编写程序文件 hello.py，其中只有一行代码：

```
C:\WINDOWS\system32\cmd.exe

      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 2.0.2
      /_/

Using Python version 3.5.2 (v3.5.2:4def2a2901a5, Jun 25 2016 22:18:55)
SparkSession available as 'spark'.
>>> import math
>>> math.sin(3)
0.1411200080598672
>>> from PIL import Image
>>>
```

图 12-8　pyspark 开发界面

```
print('Hello world')
```

然后在命令提示符环境中提交该程序即可执行，如图 12-9 所示。

```
C:\WINDOWS\system32\cmd.exe

D:\spark-2.0.2-bin-hadoop2.7\bin>spark-submit.cmd hello.py
Hello world

D:\spark-2.0.2-bin-hadoop2.7\bin>
```

图 12-9　执行 Python 程序

下面的 Python 程序文件 pi.py 用来估算圆周率的值，保存至 Spark 安装目录中的 bin 目录中，可以使用命令 spark-submit.cmd pi.py 运行程序并输出圆周率的值。

```
from pyspark import SparkConf,SparkContext
from pyspark.sql import SQLContext
from random import random

conf =SparkConf().setAppName("pi")
sc =SparkContext(conf =conf)
sqlCtx =SQLContext(sc)

def sample(p):
    x,y=random(),random()
    return 1 if x* x +y* y<1 else 0

NUM_SAMPLES =100000                              #数值越大结果越准确
count =sc.parallelize(range(NUM_SAMPLES))
count =count.map(sample).reduce(lambda a,b: a +b)
print('=' * 30)
print("Pi is roughly %f" % (4.0 *count/NUM_SAMPLES))
print('=' * 30)
```

下面的代码使用 Spark 来统计 100 000 000 以内的素数数量，在 6G RAM、双核 CPU 的 64 位 Win 7 + spark 单机平台上运行时间为 765.015 428s。



```
from pyspark import SparkConf,SparkContext

conf =SparkConf().setAppName("isPrime")
sc =SparkContext(conf =conf)
def isPrime(n):
    if n<2:
        return False
    if n==2:
        return True
    if not n&1:
        return False
    for i in range(3,int(n**0.5) +2,2):
        if n% i==0:
            return False
    return True
rdd=sc.parallelize(range(100000000))
result =rdd.filter(isPrime).count()
print('=' * 30)
print(result)
```

下面的代码在相同的平台上使用传统的方式来求解同一个问题，运行时间为1666.616 000 18s，由此可见，即使是在单机多核环境下，Spark 也会获得速度上的很大提高，提高的比例和处理器的数量有一定的关系。

```
import time

def isPrime(n):
    if n<2:
        return False
    if n==2:
        return True
    if not n&1:
        return False
    for i in range(3,int(n**0.5) +2,2):
        if n% i==0:
            return False
    return True

start =time.time()
num=sum(1 for n in range(100000000) if isPrime(n))
print(num)
print(time.time() -start)
```

下面的代码使用筛选法实现了同样的功能，但空间占用非常大，在实际应用中并不推荐。

```
from pyspark import SparkConf,SparkContext
from random import random

conf =SparkConf().setAppName("isPrime")
sc =SparkContext(conf =conf)
n =1000
m=int(n**0.5) +1
rdd=sc.parallelize(range(2,n))

result =set()
while True:
    #获取第一个元素
    t =rdd.first()
    if t >m:
        break
    result.add(t)
    #对 RDD 上的所有元素进行过滤、筛选,能被整除的全部过滤掉
    rdd=sc.parallelize(rdd.filter(lambda x: x%t !=0).collect())

print(list(result) +rdd.collect())
```

下面的代码演示了 pyspark 的很少一部分功能和用法,更加详细的函数介绍请参考网址 http://spark.apache.org/docs/latest/api/python/pyspark.html。

```
>>>from pyspark import SparkFiles
>>>path ='test.txt'
>>>with open(path,'w') as fp:                     #创建文件
      fp.write('100')
>>>sc.addFile(path)                               #提交文件
>>>def func(iterator):
      with open(SparkFiles.get('test.txt')) as fp:  #打开文件
          Val =int(fp.readline())                 #读取文件中的内容
          return [x*Val for x in iterator]
>>>sc.parallelize([1,2,3,4,5]).mapPartitions(func).collect()
                              #并行处理,collect()返回包含 RDD 上所有元素的列表
[100,200,300,400,500]
>>>sc.parallelize([2,3,4]).count()                  #count()用来返回 RDD 中元素的个数
                              #parallelize()用来分布本地的 Python 集合并创建 RDD
3
>>>rdd=sc.parallelize([1,2])
>>>sorted(rdd.cartesian(rdd).collect())           #collect()返回包含 RDD 中元素的列表
[(1,1),(1,2),(2,1),(2,2)]                         #cartesian()计算两个 RDD 的笛卡儿积
>>>rdd=sc.parallelize([1,2,3,4,5])
>>>rdd.filter(lambda x: x %2 ==0).collect()       #只保留符合条件的元素
[2,4]
```

```
>>>sorted(sc.parallelize([1,1,2,3]).distinct().collect())  #返回唯一元素
[1,2,3]
>>>rdd=sc.parallelize(range(10))
>>>rdd.map(lambda x: str(x)).collect()                #映射
>>>rdd=sc.parallelize([1.0,5.0,43.0,10.0])
>>>rdd.max()                                          #最大值
43.0
>>>rdd.max(key=str)
5.0
>>>rdd.min()                                          #最小值
1.0
>>>rdd.sum()                                          #所有元素求和
59.0
>>>from random import randint
>>>lst=[randint(1,100) for _ in range(20)]
>>>lst
[18,55,48,13,86,23,85,62,66,58,73,96,90,16,49,98,49,69,3,53]
>>>sc.parallelize(lst).top(3)                         #最大的 3 个元素
[98,96,90]
>>>sorted(lst,reverse=True)[:3]
[98,96,90]
>>>sc.parallelize(range(100)).filter(lambda x:x>90).take(3)
                                                      #使用 take()返回前 3 个元素
[91,92,93]
>>>sc.parallelize(range(20),3).glom().collect()       #查看分片情况
[[0,1,2,3,4,5],[6,7,8,9,10,11,12],[13,14,15,16,17,18,19]]
>>>sc.parallelize(range(20),6).glom().collect()       #查看分片情况
[[0,1,2],[3,4,5],[6,7,8,9],[10,11,12],[13,14,15],[16,17,18,19]]
>>>myRDD=sc.parallelize(range(20),6)                  #6 表示分片数
>>>sc.runJob(myRDD,lambda part: [x **2 for x in part]) #执行任务
[0,1,4,9,16,25,36,49,64,81,100,121,144,169,196,225,256,289,324,361]
>>>sc.runJob(myRDD,lambda part: [x **2 for x in part],[1])
                                                      #只查看第 2 个分片的结果
[9,16,25]
>>>sc.runJob(myRDD,lambda part: [x **2 for x in part],[1,5])
                                                      #查看第 2 和第 6 个分片上的结果
[9,16,25,256,289,324,361]
>>>sc.parallelize([1,2,3,3,3,2]).distinct().collect()   #distinct()返回包含唯一元素的 RDD
[1,2,3]
>>>from operator import add,mul
>>>sc.parallelize([1,2,3,4,5]).fold(0,add)            #把所有分片上的数据累加
15
>>>sc.parallelize([1,2,3,4,5]).fold(1,mul)            #把所有分片上的数据连乘
120
```

```
>>>sc.parallelize([1,2,3,4,5]).reduce(add)          #reduce()函数的并行版本
15
>>>sc.parallelize([1,2,3,4,5]).reduce(mul)
120
>>>result =sc.parallelize(range(1,6)).groupBy(lambda x: x%3).collect()
                                                     #对所有数据进行分组
>>>for k,v in result:
    print(k,sorted(v))

0 [3]
1 [1,4]
2 [2,5]
>>>rdd1 =sc.parallelize(range(10))
>>>rdd2 =sc.parallelize(range(5,20))
>>>rdd1.intersection(rdd2).collect()                 #交集
[8,9,5,6,7]
>>>rdd1.subtract(rdd2).collect()                     #差集
[0,1,2,3,4]
>>>rdd1.union(rdd2).collect()                        #合并两个 RDD 上的元素
[0,1,2,3,4,5,6,7,8,9,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19]
>>>rdd1 =sc.parallelize('abcd')
>>>rdd2 =sc.parallelize(range(4))
>>>rdd1.zip(rdd2).collect()                          #两个 RDD 必须等长
[('a',0),('b',1),('c',2),('d',3)]
>>>rdd=sc.parallelize('abcd')
>>>rdd.map(lambda x: (x,1)).collect()                #内置函数 map()的并行版本
[('a',1),('b',1),('c',1),('d',1)]
>>>sc.parallelize([1,2,3,4,5]).stdev()               #计算标准差
1.4142135623730951
>>>sc.parallelize([1,1,1,1,1]).stdev()
0.0
```

运行一下上面的代码会发现,屏幕上会出现非常详细的执行过程,实际上很多时候我们并不需要那些信息,只想关心代码的执行结果。如果想关闭这些详细信息的显示,可以把 Spark 安装文件夹的 conf 文件夹中 log4j. properties. template 文件复制一份保存到 conf 文件夹中并改名为 log4j. properties,然后使用记事本打开新文件,把里面的 INFO 都改为 WARN,关闭后重启 pyspark. cmd 就可以了,这个操作和设置对使用 spark-submit. cmd 提交并执行的程序也同样有效。

12.6 GPU 编程

自计算机诞生以来,都是由 CPU 承担着数据的计算和处理任务,著名的 NVIDIA 厂商于 2007 年做了一次大胆的尝试,使用 GPU 中功能简单而数量众多的处理模块来配合



CPU 实现并行处理，从而实现大幅度加速。这一尝试获得了巨大成功，并被迅速应用到数据挖掘与分析、并行处理、机器学习、科学计算等行业和领域。

CPU 由专为串行处理而优化的几个核心组成。GPU 由数量众多的更小、更高效的核心组成，这些核心专为同时处理多任务而设计，虽然每个核心的功能并不像 CPU 那么强大，但是用来并行处理一些小任务，还是具有很大优势的。

实现 GPU 运算首先需要显卡（例如 NVIDIA 显卡）的支持，并且根据需要安装相应的 Python 扩展库，例如 pycuda、pyopencl、theano、scikit-learn、NumbaPro、TensorFlow。不过，GPU 加速也不是万能的，并不是适用于所有场合的应用，如果需要在 CPU 和 GPU 之间频繁传输数据，反而会影响效率，不如直接使用 CPU 的速度快。

12.6.1　使用 pycuda 实现 GPU 加速

借助于扩展库 pycuda，可以在 Python 中访问 NVIDIA 显卡提供的 CUDA 并行计算 API，使用非常方便。安装 pycuda 时要求已正确安装合适版本的 CUDA 和 Visual Studio（注意，并不是版本越新越合适），然后再使用 pip 安装 pycuda。下面的案例使用 pycuda 在 GPU 上并行判断素数，测试结果显示在 640 核 GPU 上的运行速度为 4 核 CPU 的 8 倍左右。

示例 12-31　使用 pycuda 在 GPU 上并行判断素数，统计 100 000 000 之内的素数个数。

```
import time
import pycuda.autoinit
import pycuda.driver as drv
import numpy as np
from pycuda.compiler import SourceModule

#编译 C 代码进入显卡，并行判断素数
mod = SourceModule('''
__global__ void isPrime(int *dest, int *a, int *b)
{
    const int i = threadIdx.x + blockDim.x * blockIdx.x;
    int j;
    for(j = 2; j < b[i]; j++)
    {
        if(a[i] % j == 0)
        {
            break;
        }
    }
    if(j >= b[i])
    {
        dest[i] = a[i];
```

```
    }
}
''')

#定义待测数值范围,以及每次处理的数字数量
end=100000000
size=1000

#获取函数
isPrime=mod.get_function("isPrime")
result=0

start=time.time()
#分段处理,每次处理 1000 个数字
for i in range(end//size):
    startN=i*size
    a=np.array(range(startN,startN+size)).astype(np.int64)
    #b中是 a 中对应数字的平方根加 1 后的整数,用于快速判断素数
    b=np.array(list(map(lambda x: int(x**0.5)+1,a))).astype(np.int64)
    dest=np.zeros_like(a)
    isPrime(drv.Out(dest),drv.In(a),drv.In(b),
            block=(size,1,1),grid=(2,1))
    result +=len(set(filter(None,dest)))
print(time.time()-start)

#上面的代码中把 1 也算上了,这里减去
print(result-1)
```

12.6.2 使用 pyopencl 实现 GPU 加速

扩展库 pyopencl 使得可以在 Python 中调用 OpenCL 的并行计算 API。OpenCL(Open Computing Language)是跨平台的并行编程标准,可以运行在个人计算机、服务器、移动终端以及嵌入式系统等多种平台,既可以运行在 CPU 上又可以运行于 GPU 上,大幅度提高了各类应用中的数据处理速度,包括游戏、娱乐、医学软件以及科学计算等。下面的案例使用 pyopencl 在 GPU 上并行判断素数,测试结果显示在640 核 GPU 上的运行速度为 4 核 CPU 的 12 倍左右。

示例 12-32 使用 pyopencl 在 GPU 上并行判断素数,统计 100 000 000 之内的素数个数。

```
import numpy as np
import pyopencl as cl
import pyopencl.array
from pyopencl.elementwise import ElementwiseKernel
```

```
#判断素数的 C 语言版 GPU 代码
isPrime =ElementwiseKernel(ctx,
    'long *a_g,long *b_g,long *res_g',
    '''
        int j;
        for(j =2; j <b_g[i]; j ++)
        {
            if(a_g[i]%j ==0)
            {
                break;
            }
        }
        if(j  >=b_g[i])
        {
            res_g[i] =a_g[i];
        }''',
    'isPrime'
)

#定义待测数值范围,以及每次处理的数字数量
end =100000000
start_end =range(2,end)
size =1000

result =0

ctx =cl.create_some_context()
queue =cl.CommandQueue(ctx)

#对指定范围内的数字进行分批处理
for i in range(end//size  +1):
    startN =i *size
    #本次要处理的数字范围
    a_np =np.array(start_end[startN: startN +size]).astype(np.int64)
    #b_np 里的数字是 a_np 中数字的平方根取整后加 1
    b_np =np.array(list(map(lambda x: int(x**0.5) +1,a_np))).astype(np.int64)
    #把数据写入 GPU
    a_g =cl.array.to_device(queue,a_np)
    b_g =cl.array.to_device(queue,b_np)
    res_g =cl.array.zeros_like(a_g)
    #批量判断
    isPrime(a_g,b_g,res_g)
    t =set(filter(None,res_g.get()))
    #记录本批数字中素数的个数
```

```
    result +=len(t)

print(result)
```

12.6.3 使用 tensorflow 实现 GPU 加速

tensorflow 是一个用于人工智能的开源神器，是一个采用数据流图(data flow graphs)用于数值计算的开源软件库。数据流图使用节点(nodes)和边线(edges)的有向图来描述数学计算，图中的节点表示数学操作，也可以表示数据输入的起点或者数据输出的终点，而边线表示在节点之间的输入输出关系，用来运输大小可动态调整的多维数据数组，也就是张量(tensor)。tensorflow 可以在普通计算机、服务器和移动设备的 CPU 和 GPU 上展开计算，具有很强的可移植性，并且支持 C++、Python 等多种语言。

示例 12-33 使用 tensorflow 中的梯度下降算法求解变量最优值。

```
import tensorflow as tf
import numpy as np
import time

#使用 NumPy 生成随机数据，总共 2 行 100 列个点
x_data =np.float32(np.random.rand(2,200))
#矩阵乘法
#这里的 W=[0.100,0.200]和 b=0.300 是理论数据，通过后面的训练来验证
y_data =np.dot([0.100,0.200],x_data) +0.300

#构造一个线性模型，训练求解 W 和 b
#初始值 b=[0.0]
b =tf.Variable(tf.zeros([1]))
#初始值 W 为 1×2 的矩阵，元素值介于[-1.0,1.0]区间
W =tf.Variable(tf.random_uniform([1,2], -1.0,1.0))
#构建训练模型，matmul 为矩阵乘法运算
y =tf.matmul(W,x_data) +b

#最小均方差
loss =tf.reduce_mean(tf.square(y -y_data))
#使用梯度下降算法进行优化求解
optimizer =tf.train.GradientDescentOptimizer(0.5)
train =optimizer.minimize(loss)

#初始化变量
init =tf.global_variables_initializer()

with tf.device('/gpu:0'):
    with tf.Session() as sess:
```




```
#初始化
sess.run(init)

#拟合平面,训练次数越多越精确,但是也没有必要训练太多次
for step in range(0,201):
    sess.run(train)
    #显示训练过程,这里演示了两种查看变量值的方法
    print(step,sess.run(W),b.eval())
```

第 13 章 互通互联：asyncio 提供的网络通信功能

关于 Socket 编程的内容，在作者的另一本书《Python 可以这样学》（清华大学出版社，书号为 9787302456469）里介绍了很多，为了避免重复，已经介绍过的基础知识和案例就不在本书出现了，本书把重点放在标准库 asyncio 提供的网络通信功能上。另外，在第 12 章介绍多进程编程的案例时也涉及了网络通信功能，请自行翻阅。

13.1 Transport 类与 Protocol 类

标准库 asyncio 提供的 BaseTransport、ReadTransport、WriteTransport、DatagramTransport 以及 BaseSubprocessTransport 类对不同类型的信道进行了抽象。一般来说不要使用这些类去直接实例化对象，而是应该调用 AbstractEventLoop 函数来创建相应的 Transport 对象并且对底层信道进行初始化。一旦信道创建成功，就可以通过一对 Protocol 对象进行通信了。目前 asyncio 支持 TCP、UDP、SSL 和 Subprocess 管道，不同类型的 Transport 对象支持的方法略有不同。另外需要注意的是，Transport 类不是线程安全的。

标准库 asyncio 还提供了类 Protocol、DatagramProtocol 和 SubprocessProtocol，这些类可用作基类进行二次开发来实现自己的网络协议，创建派生类时只需重写感兴趣的回调函数即可，详见表 13-1。Protocol 类常与 Transport 类一起使用，Protocol 对象解析收到的数据并请求待发出数据的写操作，而 Transport 对象则负责实际的 I/O 操作和必要的缓冲。

表 13-1　Protocol 对象常用回调函数

函数名称	说　　明	适用对象
connection_made(transport)	连接建立后自动调用	Protocol Datagram-Protocol SubProcessProtocol
connection_lost(exc)	连接丢失或关闭后自动调用	
pipe_data_received(fd, data)	子进程往 stdout 或 stderr 管道中写入数据时自动调用，fd 是管道的标识符，data 是要写入的非空字节串	SubProcessProtocol
pipe_connection_lost(fd, exc)	与子进程通信的管道被关闭时自动调用	
process_exited()	子进程退出后自动调用	

续表

函 数 名 称	说　　明	适用对象
data_received(data)	收到数据(字节串)时自动调用	Protocol
eof_received()	通信对方通过 write_eof()或其他类似方法通知不再发送数据时自动调用	
datagram_received(data, addr)	收到数据报时自动调用	DatagramProtocol
error_received(exc)	前一次发送或接收操作抛出异常 OSError 时自动调用	
pause_writing()	Transport 对象缓冲区达到上水位线时自动调用	Protocol DatagramProtocol SubProcessProtocol
resume_writing()	Transport 对象缓冲区达到下水位线时自动调用	

可以在 Protocol 对象的方法中使用 ensure_future()来启动协程,但并不保证严格的执行顺序,Protocol 对象并不清楚在对象方法中创建的协程,所以也不会等待其执行结束。如果需要确定执行顺序的话,可以在协程中通过 yield from 语句来使用 Stream 对象。

示例 13-1　使用 TCP 进行通信。

(1) 服务端代码。

```
import asyncio

class EchoServerClientProtocol(asyncio.Protocol):
    #连接建立成功
    def connection_made(self,transport):
        peername =transport.get_extra_info('peername')
        print('Connection from {}'.format(peername))
        self.transport =transport

    #收到数据
    def data_received(self,data):
        message =data.decode()
        print('Data received: {!r}'.format(message))

        print('Send: {!r}'.format(message))
        self.transport.write(data)

    #对方发送消息结束
    def eof_received(self):
        print('Close the client socket')
        self.transport.close()

loop =asyncio.get_event_loop()
#创建服务器,每个客户端的连接请求都会创建一个新的 Protocol 实例
```

```
coro=loop.create_server(EchoServerClientProtocol,'127.0.0.1',8888)
server=loop.run_until_complete(coro)

#服务器一直运行,直到用户按下 Ctrl+C 键
print('Serving on {}'.format(server.sockets[0].getsockname()))
try:
    loop.run_forever()
except KeyboardInterrupt:
    pass

#关闭服务器
server.close()
loop.run_until_complete(server.wait_closed())
loop.close()
```

（2）客户端代码。

```
import asyncio
import time

class EchoClientProtocol(asyncio.Protocol):
    def __init__(self,message,loop):
        self.message=message
        self.loop=loop

    #连接创建成功
    def connection_made(self,transport):
        for m in message:
            transport.write(m.encode())
            print('Data sent: {!r}'.format(m))
            time.sleep(1)
        #全部消息发送完成,通知对方不再发送消息
        transport.write_eof()

    #收到数据
    def data_received(self,data):
        print('Data received: {!r}'.format(data.decode()))

    #连接被关闭
    def connection_lost(self,exc):
        print('The server closed the connection')
        print('Stop the event loop')
        self.loop.stop()

loop=asyncio.get_event_loop()
```

```
message=['Hello World!','你好']
coro=loop.create_connection(lambda: EchoClientProtocol(message,loop),
                            '127.0.0.1',8888)
loop.run_until_complete(coro)
loop.run_forever()
loop.close()
```

示例 13-2　使用 UDP 进行通信，模拟时间服务器，服务端可以接收客户端的定期查询并返回当前时间。

（1）监听端代码。

```
import asyncio
import datetime
import socket

class EchoServerProtocol:
    def connection_made(self,transport):
        self.transport=transport

    def datagram_received(self,data,addr):
        message=data.decode()
        print('Received from',str(addr))
        now=str(datetime.datetime.now())[:19]
        self.transport.sendto(now.encode(),addr)
        print('replied')

loop=asyncio.get_event_loop()
print("Starting UDP server")
#获取本机 IP 地址
ip=socket.gethostbyname(socket.gethostname())
#创建 Protocol 实例,服务所有客户端
listen=loop.create_datagram_endpoint(EchoServerProtocol,
                                     local_addr=(ip,9999))
transport,protocol=loop.run_until_complete(listen)

try:
    loop.run_forever()
except KeyboardInterrupt:
    pass

transport.close()
loop.close()
```

（2）客户端代码。

```
import asyncio
import time

class EchoClientProtocol:
    def __init__(self,message,loop):
        self.message =message
        self.loop =loop

    def connection_made(self,transport):
        self.transport =transport
        self.transport.sendto(self.message.encode())

    def datagram_received(self,data,addr):
        print("Now is:",data.decode())
        self.transport.close()

    def error_received(self,exc):
        print('Error received:',exc)

    def connection_lost(self,exc):
        self.loop.stop()

loop =asyncio.get_event_loop()
message ="ask for time"
while True:
    connect =loop.create_datagram_endpoint(
                                    lambda: EchoClientProtocol(message,
                                    loop),remote_addr =('10.2.1.2',9999))
    transport,protocol =loop.run_until_complete(connect)
    loop.run_forever()
    transport.close()
    time.sleep(1)
loop.close()
```

示例 13-3 注册用于接收数据的 Socket,并实现两个 Socket 之间的数据传输。

```
import asyncio

try:
    from socket import socketpair
except ImportError:
    from asyncio.windows_utils import socketpair

class MyProtocol(asyncio.Protocol):
    def connection_made(self,transport):
```




```
        self.transport = transport

    def data_received(self, data):
        #接收数据,关闭 Transport 对象
        print("Received:", data.decode())
        self.transport.close()

    def connection_lost(self, exc):
        #Socket 已被关闭,停止事件循环
        loop.stop()

#创建一对互相连通的 Socket
rsock, wsock = socketpair()
loop = asyncio.get_event_loop()

#注册用来等待接收数据的 Socket
connect_coro = loop.create_connection(MyProtocol, sock = rsock)
transport, protocol = loop.run_until_complete(connect_coro)

#往互相连通的 Socket 中的一个写入数据
loop.call_soon(wsock.send, 'hello world.'.encode())

#启动事件循环
loop.run_forever()

rsock.close()
wsock.close()
loop.close()
```

13.2　StreamReader 与 StreamWriter

asyncio 模块还提供了 open_connection()函数(对 AbstractEventLoop.create_connection()函数的封装)、open_unix_connection()函数(对 AbstractEventLoop.create_unix_connection()函数的封装)以及 start_server()(对 AbstractEventLoop.create_server()函数的封装)和 start_unix_server()函数,这些都是协程函数,其中的参数含义与被封装的函数基本一致。

open_connection()函数执行成功的话会返回(reader, writer),其中 reader 是 StreamReader 类的实例,而 writer 是 StreamWriter 类的实例。StreamReader 类提供了 set_transport(transport)、feed_data(data)和 feed_eof()方法以及协程方法 read(n = -1)、readline()、readexactly(n)、readuntil(separator = b'\n')用来从 Transport 对象中读取数据;封装了 Transport 类的 StreamWriter 类则提供了普通方法 close()、get_extra_info()、write(data)、writelines(data)、write_eof()和协程方法 drain()(如果 Transport 对象的缓冲区达

到上水位线就会阻塞写操作，直到缓冲区大小被拉到下水位线时再恢复）。

示例 13-4 使用 StreamReader 和 StreamWriter 实现网络聊天程序。

（1）服务端代码。

```
import asyncio

messages ={'Hello':'nihao',
           'How are you?':'Fine,thank you.',
           'Did you have breakfast?':'Yes',
           'Bye':'Bye'}
@asyncio.coroutine
def handle_echo(reader,writer):
    while True:
        data =yield from reader.read(100)
        message =data.decode()
        addr =writer.get_extra_info('peername')
        print("Received %r from %r" % (message,addr))

        messageReply =messages.get(message,'Sorry')
        print("Send: %r" %messageReply)
        writer.write(messageReply.encode())
        yield from writer.drain()
        if messageReply =='Bye':
            break

    print("Close the client socket")
    writer.close()

#创建事件循环
loop =asyncio.get_event_loop()
#创建并启动服务器
coro =asyncio.start_server(handle_echo,'10.2.1.2',8888,loop =loop)
server =loop.run_until_complete(coro)

print('Serving on {}'.format(server.sockets[0].getsockname()))
#按 Ctrl +C键或 Ctrl +Break 键退出
try:
    loop.run_forever()
except KeyboardInterrupt:
    pass

#关闭服务器
server.close()
loop.run_until_complete(server.wait_closed())
```

```
loop.close()
```

（2）客户端代码。

```
import asyncio

@asyncio.coroutine
def tcp_echo_client(loop):
    reader,writer =yield from asyncio.open_connection('10.2.1.2',
                                                      8888,loop =loop)

    while True:
        message =input('You said:')
        writer.write(message.encode())
        data =yield from reader.read(100)
        print('Received: %r' %data.decode())
        if message =='Bye':
            break

    print('Close the socket')
    writer.close()

loop =asyncio.get_event_loop()
loop.run_until_complete(tcp_echo_client(loop))
loop.close()
```

示例 13-5　使用 StreamReader 和 StreamWriter 获取网页头部信息。

```
import asyncio
import urllib.parse
import sys

@asyncio.coroutine
def print_http_headers(url):
    url =urllib.parse.urlsplit(url)
    if url.scheme =='https':
        connect =asyncio.open_connection(url.hostname,443,ssl =True)
    else:
        connect =asyncio.open_connection(url.hostname,80)
    reader,writer =yield from connect

    query =('HEAD {path} HTTP/1.0\r\nHost: {hostname}\r\n\r\n'
            ).format(path =url.path or '/',hostname =url.hostname)
    writer.write(query.encode('latin -1'))

    while True:
```

```
            line =yield from reader.readline()
            if not line:
                break
            line =line.decode('latin1').rstrip()
            if line:
                print('HTTP header >%s' % line)

    writer.close()

url ='https://docs.python.org/3/library/asyncio-stream.html'
loop =asyncio.get_event_loop()
task =asyncio.ensure_future(print_http_headers(url))
loop.run_until_complete(task)
loop.close()
```

示例 13-6 注册端口并接收数据。

```
import asyncio

try:
    from socket import socketpair
except ImportError:
    from asyncio.windows_utils import socketpair

@asyncio.coroutine
def wait_for_data(loop):
    #创建一对互相连通的 Socket
    rsock,wsock =socketpair()

    #注册用来接收数据的 Socket
    reader,writer =yield from asyncio.open_connection(sock =rsock,loop =loop)

    #通过 Socket 写入数据
    loop.call_soon(wsock.send,'This is a test.'.encode())

    #等待接收数据
    data =yield from reader.read(100)
    print("Received:",data.decode())

    writer.close()
    wsock.close()

loop =asyncio.get_event_loop()
loop.run_until_complete(wait_for_data(loop))
loop.close()
```

附录



精彩在继续

附录 A　GUI 开发

目前适用于 Python 的 GUI 库主要有 wxPython、tkinter 和 PyQT，其中 wxPython 似乎更强大一些，不过需要额外安装才行，虽然这并不麻烦。而 tkinter 是 Python 标准库，可以直接使用，也具有很大的优势。关于 wxPython 的更多介绍请参考我的另外一本书《Python 程序设计(第 2 版)》(清华大学出版社，书号为 9787302436515)，tkinter 的大量案例代码可以参考我的另外一本书《Python 可以这样学》(清华大学出版社，书号为 9787302456469)。这里再通过一个 GUI 版的猜数游戏介绍一下 tkinter 的高级用法。

把下面的代码保存并运行之后，首先需要启动游戏并设置数值范围和最大允许猜数次数，然后才能在文本框内输入猜测的数字，程序会提示正确、数值过大或过小，玩家根据提示对下一次猜数进行调整，超过次数限制之后游戏结束并提示正确的数字，退出程序时提示战绩。游戏运行初始界面如附图 A.1 所示。



图 A.1　猜数游戏运行初始界面

```
import random
import tkinter
import tkinter.messagebox
import tkinter.simpledialog

root = tkinter.Tk()
#窗口标题
root.title('猜数游戏——by 董付国')
#窗口初始大小和位置
root.geometry('280x80 +400 +300')
#不允许改变窗口大小
root.resizable(False, False)

#用户猜的数
varNumber = tkinter.StringVar(root, value = '0')
#允许猜的总次数
totalTimes = tkinter.IntVar(root, value = 0)
```

```
#已猜次数
already=tkinter.IntVar(root,value=0)
#当前生成的随机数
currentNumber=tkinter.IntVar(root,value=0)
#玩家玩游戏的总次数
times=tkinter.IntVar(root,value=0)
#玩家猜对的总次数
right=tkinter.IntVar(root,value=0)

lb=tkinter.Label(root,text='请输入一个整数：')
lb.place(x=10,y=10,width=100,height=20)
#用户猜数并输入的文本框
entryNumber=tkinter.Entry(root,width=140,textvariable=varNumber)
entryNumber.place(x=110,y=10,width=140,height=20)
#只有开始游戏以后才允许输入
entryNumber['state']='disabled'

#关闭程序时提示战绩
def closeWindow():
    message='共玩游戏 {0} 次，猜对 {1} 次！ \n欢迎下次再玩！ '
    message=message.format(times.get(),right.get())
    tkinter.messagebox.showinfo('战绩',message)
    root.destroy()
root.protocol('WM_DELETE_WINDOW',closeWindow)

#按钮单击事件处理函数
def buttonClick():
    if button['text']=='Start Game':
        #每次游戏时允许用户自定义数值范围
        #玩家必须输入正确的数
        while True:
            try:
                start=tkinter.simpledialog.askinteger('允许的最小整数',
                                                  '最小数',initialvalue=1)
                break
            except:
                pass
        while True:
            try:
                end=tkinter.simpledialog.askinteger('允许的最大整数',
                                                '最大数',initialvalue=10)
                break
            except:
                pass
```

```
        #在用户自定义的数值范围内生成随机数
        currentNumber.set(random.randint(start,end))
        #用户自定义一共允许猜几次
        #玩家必须输入正确的整数
        while True:
            try:
                t=tkinter.simpledialog.askinteger('最多允许猜几次?',
                                                  '总次数',initialvalue=3)
                totalTimes.set(t)
                break
            except:
                pass
        #已猜次数初始化为 0
        already.set(0)
        button['text']='剩余次数: ' +str(t)
        #把文本框初始化为 0
        varNumber.set('0')
        #允许用户开始输入整数
        entryNumber['state']='normal'
        #玩游戏的次数加 1
        times.set(times.get() +1)
    else:
        #一共允许猜几次
        total=totalTimes.get()
        #本次游戏的正确答案
        current=currentNumber.get()
        #玩家本次猜的数
        try:
            x=int(varNumber.get())
        except:
            tkinter.messagebox.showerror('抱歉','必须输入整数')
            return
        if x==current:
            tkinter.messagebox.showinfo('恭喜','猜对了')
            button['text']='Start Game'
            #禁用文本框
            entryNumber['state']='disabled'
            right.set(right.get() +1)
        else:
            #已猜次数加 1
            already.set(already.get()+1)
            if x>current:
                tkinter.messagebox.showerror('抱歉','猜的数太大了')
            else:
```

```
            tkinter.messagebox.showerror('抱歉','猜的数太小了')
        #可猜次数用完了
        if already.get() ==total:
            tkinter.messagebox.showerror('抱歉',
                                         '游戏结束了,正确的数是:' +
                                         str(currentNumber.get()))
            button['text'] ='Start Game'
            #禁用文本框
            entryNumber['state'] ='disabled'
        else:
            button['text'] ='剩余次数:' +str(total -already.get())
#在窗口上创建按钮,并设置事件处理函数
button =tkinter.Button(root,text ='Start Game',command =buttonClick)
button.place(x =10,y =40,width =250,height =20)

#启动消息主循环
root.mainloop()
```

上面的小游戏是使用代码生成的 tkinter 界面,这样做对只有少量组件的界面是没有问题的,但是如果界面上需要放置大量的组件,这样手动创建就比较费时费力了,可以考虑使用 PAGE 来实现复杂的界面设计。

安装好并启动 PAGE 之后,首先创建一个 Toplevel,然后在左侧 Widget Toolbar 中选择需要创建的组件,在刚刚创建的 Toplevel 中合适位置单击即可创建组件,将其拖放至合适的位置,最后在右侧的 Attribute Editor 设置组件的属性和有关的操作。界面设计好以后,单击菜单 Gen_Python 中的子菜单 Generate Python GUI 生成界面程序,再使用菜单 Generate Support Module 生成 Python 程序文件,填写必要的命令处理(如单击按钮)代码后保存即可。

附录 B　SQLite 数据库操作

SQLite 是内嵌在 Python 中的轻量级、基于磁盘文件的数据库管理系统,**不需要安装和配置服务器**,支持使用 SQL 语句来访问数据库。该数据库使用 C 语言开发,支持大多数 SQL91 标准,支持原子的、一致的、独立的和持久的事务,不支持外键限制;通过数据库级的独占性和共享锁定来实现独立事务,当多个线程同时访问同一个数据库并试图写入数据时,每一时刻只有一个线程可以写入数据。

SQLite 支持最大 140TB 大小的单个数据库,**每个数据库完全存储在单个磁盘文件中**,以 B^+ 树数据结构的形式存储,一个数据库就是一个文件,通过直接复制数据库文件就可以实现备份。使用 SQLite 数据库并不需要专门启动什么服务,也没有开放访问端口,这意味着不能访问其他机器上的 SQLite 数据库,很难把程序服务器和数据库服务器分离,当然可以自己编写相应的通信程序来解决这个问题。如果需要使用可视化管理工具,可以使用 SQLiteManager、SQLite Database Browser 或其他类似工具。

Python 标准库 sqlite3 提供了 SQLite 数据库访问接口，连接数据库之后剩下的工作就是使用 SQL 语句对数据进行增、删、改、查了。能够阅读本书的读者相比已经有了一些 SQL 语句的编写经验，就不再过多介绍相关语法了。下面的代码简单演示了 sqlite3 模块的用法。

```
>>>import sqlite3
>>>conn =sqlite3.connect('test.db')                    #连接或创建数据库
>>>cur =conn.cursor()                                   #创建游标
>>>cur.execute('CREATE TABLE tableTest(field1 numeric,field2 text)')
                                                        #创建数据表
<sqlite3.Cursor object at 0x000001C7AB3B43B0 >
>>>data =zip(range(5),'abcde')
>>>cur.executemany('INSERT INTO tableTest values(?,?)',data)
                                                        #插入多条记录
<sqlite3.Cursor object at 0x000001C7AB3B43B0 >
>>>cur.execute('SELECT *FROM tableTest ORDER BY field1 DESC')
                                                        #查询记录
<sqlite3.Cursor object at 0x000001C7AB3B43B0 >
>>>for rec in cur.fetchall():
  print(rec)

(4,'e')
(3,'d')
(2,'c')
(1,'b')
(0,'a')
```

附录 C　计算机图形学编程

计算机图形学主要研究如何使用计算机来生成具有真实感的图形，涉及的内容主要包括三维建模、图形几何变换、光照模型、纹理映射、阴影模型等内容，在机械制造、虚拟现实、增强现实、游戏开发、漫游系统设计、产品展示等多个领域具有重要的应用。随着 3D 打印机的诞生，只要有模型就能够快速生成实物，无疑这将会大大扩展计算机图形学的应用范围，例如，可以使用计算机图形学的技术制作出各种可爱的模型，然后参照这些模型使用 3D 打印机批量生产各种食品、玩偶、饰品和人体器官等。目前大部分计算机图形学的书籍都是基于 OpenGL 的，Python 也提供了相应的扩展库 pyopengl，提供了图形学编程所需要的所有 API 函数，极大方便了编写图形学程序的 Python 程序员。下面的代码使用 OpenGL 绘制了一个茶壶，实现了基本的材质和光照模型，并支持缩放和绕不同坐标轴的旋转操作。

```
import sys
from OpenGL.GL import *
```

```
from OpenGL.GLUT import *
from OpenGL.GLU import *

class MyPyOpenGLTest:
    def __init__(self,width=640,height=480,title=b'SolidTeapot'):
        glutInit(sys.argv)
        glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH)
        glutInitWindowSize(width,height)
        self.window=glutCreateWindow(title)
        glutDisplayFunc(self.Draw)
        #指定键盘事件处理函数
        glutKeyboardFunc(self.KeyPress)
        glutIdleFunc(self.Draw)
        self.InitGL(width,height)
        #绕各坐标轴旋转的角度
        self.x=0.0
        self.y=0.0
        self.z=0.0
        #缩放比例
        self.s=1.0

    def KeyPress(self,key,x,y):
        #根据不同的按键决定缩放比例和每个轴的旋转角度
        if key==b'a':
            self.x +=1
        elif key==b's':
            self.x -=1
        elif key==b'j':
            self.y +=1
        elif key==b'k':
            self.y -=1
        elif key==b'g':
            self.z +=1
        elif key==b'h':
            self.z -=1
        elif key==b'x':
            self.s +=0.3
        elif key==b'w':
            self.s -=0.3

    #绘制图形
    def Draw(self):
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
        glLoadIdentity()
```




```
        #平移
        glTranslatef(0.0,0.0,-8.0)

        #分别绕 x、y、z 轴旋转
        glRotatef(self.x,1.0,0.0,0.0)
        glRotatef(self.y,0.0,1.0,0.0)
        glRotatef(self.z,0.0,0.0,1.0)

        #各方向等比例缩放
        glScalef(self.s,self.s,self.s)

        #绘制茶壶
        #glColor3f(0.8,0.3,1.0)
        glutSolidTeapot(1.0)

        glutSwapBuffers()

    def InitGL(self,width,height):
        #初始化窗口背景为白色
        glClearColor(1.0,1.0,1.0,0.0)
        glClearDepth(1.0)
        glDepthFunc(GL_LESS)
        #设置材质与光源属性
        mat_sp=(1.0,1.0,1.0,1.0)
        mat_sh=[50.0]
        light_position=(-0.5,1.5,1,0)
        yellow_l=(1,0.7,0,1)
        ambient=(0.1,0.8,0.2,1.0)
        glMaterialfv(GL_FRONT,GL_SPECULAR,mat_sp)
        glMaterialfv(GL_FRONT,GL_SHININESS,mat_sh)
        glLightfv(GL_LIGHT0,GL_POSITION,light_position)
        glLightfv(GL_LIGHT0,GL_DIFFUSE,yellow_l)
        glLightfv(GL_LIGHT0,GL_SPECULAR,yellow_l)
        glLightModelfv(GL_LIGHT_MODEL_AMBIENT,ambient)
        #启用光照模型
        glEnable(GL_LIGHTING)
        glEnable(GL_LIGHT0)
        glEnable(GL_DEPTH_TEST)
        #光滑渲染
        glEnable(GL_BLEND)
        glShadeModel(GL_SMOOTH)
        glEnable(GL_POINT_SMOOTH)
        glEnable(GL_LINE_SMOOTH)
        glEnable(GL_POLYGON_SMOOTH)
```

```
        glMatrixMode(GL_PROJECTION)
        #反走样,也称为抗锯齿
        glHint(GL_POINT_SMOOTH_HINT,GL_NICEST)
        glHint(GL_LINE_SMOOTH_HINT,GL_NICEST)
        glHint(GL_POLYGON_SMOOTH_HINT,GL_FASTEST)
        glLoadIdentity()
        #透视投影变换
        gluPerspective(45.0,float(width)/float(height),0.1,100.0)
        glMatrixMode(GL_MODELVIEW)

    def MainLoop(self):
        glutMainLoop()

if __name__=='__main__':
    w=MyPyOpenGLTest()
    w.MainLoop()
```

附录 D　图 像 编 程

Python 扩展库 pillow 提供了非常强大的图像处理有关的功能,支持 BMP、PNG、JPEG、GIF 等多种图像格式。该扩展库主要提供了 Image、ImageChops、ImageColor、ImageDraw、ImagePath、ImageFile、ImageGrab、ImageTk、ImageEnhance、PSDraw 以及其他一些模块来支持图像处理有关的操作,而 ImageGrab 模块还支持对屏幕指定区域进行截图。在我的另外一本书《Python 可以这样学》(清华大学出版社,书号为 9787302456469)中介绍了大量的 pillow 应用,这里就不再重复了,再给出两个新的案例。下面的代码可以批量为指定文件夹中所有图像添加数字水印。

```
from random import randint
from os import listdir
from PIL import Image

#打开并读取其中的水印像素,也就是那些不是白色背景的像素
#读到内存中,放到字典中以供快速访问
im=Image.open('watermark.bmp')
width,height=im.size
pixels=dict()

for w in range(width):
    for h in range(height):
        c=im.getpixel((w,h))[:3]
        if c!=(255,255,255):
            pixels[(w,h)]=c
```



```
def addWaterMark(srcDir):
    #获取当前所有 BMP 图像文件列表
    picFiles =[fn for fn in listdir(srcDir) if fn.endswith(('.bmp','.jpg','.png'))]
    #遍历所有文件,为每个图像添加水印
    for fn in picFiles:
        im1 =Image.open(fn)
        w,h =im1.size
        #如果图片尺寸小于水印图片,不加水印
        if w <width or h <height:
            continue
        #在原始图像左上角、中间或右下角添加数字水印
        #具体位置根据 position 进行随机选择
        p ={0:(0,0),1:((w -width)//2,(h -height)//2),2:(w -width,h -height)}
        position =randint(0,2)
        top,left =p.get(position,(0,0))
        #修改像素值,添加水印
        for p,c in pixels.items():
            im1.putpixel((p[0] +top,p[1] +left),c)
        #保存加入水印之后的新图像文件
        im1.save(fn[:-4] +'_new' +fn[-4:])

#为当前文件夹中的图像文件添加水印
addWaterMark('.')
```

下面的代码用来把水印信息打散后添加到图像中的随机位置,并可以重新把水印信息提取出来。

```
from os import remove
from os.path import isfile
from random import sample,choice
from PIL import Image

def mergeWaterMark(originPic,watermarkPic,logTxt):
    #原始图片和水印文件必须为图片格式
    if ((not originPic.endswith(('.jpg','.bmp','.png'))) or
        (not watermarkPic.endswith(('.jpg','.bmp','.png')))):
        return 'Error format.'

    #打开原图和水印图片,并获取大小
    imOrigin =Image.open(originPic)
    originWidth,originHeight =imOrigin.size
    imWaterMark =Image.open(watermarkPic)
    watermarkWidth,watermarkHeight =imWaterMark.size

    #随机生成水印位置
```

```
    allPositions=[(w,h) for w in range(originWidth) for h in range(originHeight)]
    positions=sample(allPositions,watermarkWidth* watermarkHeight)

    fpLog=open(logTxt,'w')
    #写入水印文件大小
    fpLog.write(str((watermarkWidth,watermarkHeight))+'\n')

    for w in range(watermarkWidth):
        for h in range(watermarkHeight):
            c=imWaterMark.getpixel((w,h))
            c=c[:3]
            #只写入不是白色的像素
            if c!=(255,255,255):
                p=choice(positions)
                #写入像素值
                imOrigin.putpixel(p,c)
                #避免重复修改同一个像素
                positions.remove(p)
                #生成日志文件,用来提取水印
                fpLog.write(str(p+(w,h))+'\n')
    fpLog.close()
    #生成加入水印的新图片
    imOrigin.save(originPic[:-4]+'_new'+originPic[-4:])

def restoreWaterMark(mergedPic,logTxt,watermarkPic):
    #首先删除原来提取过的水印文件
    if isfile(watermarkPic):
        remove(watermarkPic)
    imMerged=Image.open(mergedPic)
    with open(logTxt) as fp:
        for line in fp:
            #读取每一行并还原为元组
            line=eval(line.strip())
            #第一行是水印图片尺寸,先创建水印文件
            if len(line)==2:
                imWaterMark=Image.new('RGB',line,(255,255,255))
            else:
                #提取水印像素并写入水印文件
                c=imMerged.getpixel((line[0],line[1]))
                c=c[:3]
                imWaterMark.putpixel((line[2],line[3]),c)
    #保存提取的水印
    imWaterMark.save(watermarkPic)
```



```
#添加水印
mergeWaterMark('origin.bmp','watermark.png','logg.txt')
#提取水印
restoreWaterMark('origin_new.bmp','logg.txt','restoredWaterMark.png')
```

附录 E　数据分析、机器学习、科学计算可视化

用于数据分析与科学计算可视化的 Python 模块非常多，例如 numpy、scipy、pandas、statistics、matplotlib、sympy、traits、traitsUI、Chaco、TVTK、Mayavi、VPython、OpenCV。其中，numpy 模块是科学计算包，提供了 Python 中没有的数组对象，支持 N 维数组运算、处理大型矩阵、成熟的广播函数库、矢量运算、线性代数、傅里叶变换以及随机数生成等功能，可与 C++、FORTRAN 等语言无缝结合，树莓派 Python v3 默认安装就已包含了 numpy。scipy 模块依赖于 numpy，提供了更多的数学工具，包括矩阵运算、线性方程组求解、积分、优化等。matplotlib 是比较常用的绘图模块，可以快速地将各种计算结果以各种图形形式展示出来。OpenCV 是一个实时的计算机视觉工具包，提供了大量相关的功能，已经成为事实上的标准工具。

近几年来，大数据、机器学习、深度学习等领域推出了大量并行计算和利用 GPU 加速的模块，例如 pycuda、pyopencl、theano、scikit-learn、NumbaPro、pySpark、TensorFlow。除了本书第 12 章中关于 pySpark 和 GPU 并行计算的介绍，在作者的另一本书《Python 可以这样学》（书号：9787302456469）和微信公众号“Python 小屋”也有一些科学计算可视化的案例代码可以参考。

Python 的大部分扩展库都可以使用 pip 命令直接安装，如果有不能安装或者安装之后无法正常工作的扩展库，可以登录下面的网页选择合适的版本下载和安装：

```
http://www.lfd.uci.edu/~gohlke/pythonlibs/
```

附录 F　密码学编程

除了自己设计加密算法或者自己编写程序实现经典的加密解密算法之外，还可以充分利用 Python 标准库和扩展库提供的丰富功能。Python 标准库 hashlib 实现了 SHA1、SHA224、SHA256、SHA384、SHA512 以及 MD5 等多个安全哈希算法，标准库 zlib 提供了 adler32 和 crc32 算法的实现，标准库 hmac 实现了 HMAC 算法。在众多的 Python 扩展库中，pycryptodome 可以说是密码学编程模块中最成功也是最成熟的一个，封装了密码学有关的大量算法，具有很高的市场占有率。另外，cryptography 也有一定数量的用户在使用。扩展库 pycryptodome 和 cryptography 提供了 SHA 系列算法和 RIPEMD160 等多个安全哈希算法，以及 DES、AES、RSA、DSA、ElGamal 等多个加密算法和数字签名算法的实现。

附录 G　系 统 运 维

系统运维涉及的内容非常广泛，例如内存、CPU、网络带宽等资源占用率以及磁盘配

额情况的实时查看，进程列表、线程列表以及用户文件变化情况的动态跟踪，网络主机 IP 地址的动态分配与回收以及 DNS 管理，用户文件变化情况，病毒防护与入侵检测，必要的时候给系统管理员发送邮件，历史数据永久化以及相关图表生成，等等。当然，严格来说，保持供电和供水系统的正常工作也属于系统运维的范畴，不过这些内容不在本书讨论范围之内。在编写系统运维程序时常用的 Python 标准库和扩展库如下。

（1）difflib：可以比较文件差异并可以生成不同格式的比较结果。

（2）filecmp：用于实现文件与文件夹的差异比较。

（3）smtplib、poplib、ftplib：邮件收发与 FTP 空间访问。

（4）ansible-playbook：轻量级多主机部署与配置管理系统。

（5）dnspython：DNS 工具包，支持几乎所有记录类型。

（6）ipy：用于管理 IPv4 和 IPv6 地址与网络的工具包。

（7）paramiko：提供了 SSHv2 协议的服务端和客户端功能。

（8）psutil：可以获取内存、CPU、磁盘、网络的使用情况，查看系统进程与线程信息，并具有一定的进程和线程管理功能。

（9）pyclamad：提供了免费开源杀毒软件 Clam Antivirus 的访问接口。

（10）pycurl：对 libcurl 的封装，类似于标准库 urllib，但功能更强大。

（11）python-rrdtool：提供了 rddtool 的访问接口，rddtool 主要用来跟踪对象的变化情况并生成走势图，例如业务的访问流量、系统性能、磁盘利用率等趋势图。

（12）scapy：交互式数据包处理工具包，支持各种网络数据包的解析和伪造。

（13）xlrd、xlwt、openpyxl：支持不同版本 Excel 文件的读写操作，包括数字、文本、公式、图表。

附录 H　Windows 系统编程

绝大多数版本的 Linux 系统中都内置了 Python 解释器，而在 Windows 平台上一般需要单独安装，尽管如此，Python 在 Windows 平台上的表现也是非常不俗的，绝大部分标准库中的功能都能在 Windows 平台上使用，并且还拥有大量专门针对 Windows 的扩展库。

（1）ctypes：Python 标准库 ctypes 提供了访问. dll 或. so 等不同类型动态链接库中函数的接口，很好地支持了与 C/C ++ 等语言混合编程的需求，可以调用操作系统底层 API 函数。

（2）os：可以调用 Windows 内部命令和外部程序，提供了一定的文件与文件夹管理功能以及进程管理功能。

（3）platform：扩平台的标准库，实现了与系统平台有关的部分功能，例如查看机型、CPU、操作系统类型等信息。

（4）winreg：提供了用于操作 Windows 系统注册表的大部分功能。

（5）wmi：提供了 Windows Management Instrumentation（WMI）的访问接口。

（6）py2exe：可以用来把 Python 程序打包成可以脱离 Python 解释器环境并独立运行在 Windows 平台上的可执行程序。

（7）pywin32：包括 win32api、win32process、win32api、win32con、win32gui、win32evtlog、win32security、winerror 等大量模块，对 Windows 底层 API 进行了完美的封装，几乎支持 Windows 平台上的所有操作。

为了方便教学，作者使用 Python 开发了一套教学管理软件，具有在线点名、提问、答疑、交作业、自测、在线考试、数据导入导出与汇总、Word 试卷生成等多个功能，其中在线考试系统具有防作弊的功能，不少人觉得很神奇，其实思路和代码都很简单。主要的原理是关闭文本编辑器并定时清空系统剪贴板，不允许复制题目和其他任何内容，也不允许打开浏览器搜索网页，只能一个题一个题地做，并且每个人都是随机抽题，题库里有 700 多道题，所以相邻的两个人同一时间抽到同一题的概率非常小，有效防止了作弊。下面的代码模拟了这个功能，单击"开始考试"按钮启用考试模式的防作弊功能，单击"结束考试"则禁用防作弊功能。代码主要使用了标准库 ctypes 和扩展库 psutil 中的部分功能。

```
import os
import time
import tkinter
import threading
import ctypes
import psutil

root = tkinter.Tk()
root.title('防作弊演示——by 董付国')
#窗口初始大小和位置
root.geometry('250x80 +300 +100')
#不允许改变窗口大小
root.resizable(False,False)
jinyong = tkinter.IntVar(root,0)

def funcJinyong():
    while jinyong.get() == 1:
        #强行关闭主流文本编辑器和网页浏览器
        for pid in psutil.pids():
            try:
                p = psutil.Process(pid)
                exeName = os.path.basename(p.exe()).lower()
                if exeName in ('notepad.exe','winword.exe',
                               'wps.exe','wordpad.exe','iexplore.exe',
                               'chrome.exe','qqbrowser.exe',
                               '360chrome.exe','360se.exe',
                               'sogouexplorer.exe','firefox.exe',
                               'opera.exe','maxthon.exe',
                               'netscape.exe','baidubrowser.exe',
                               '2345Explorer.exe'):
                    p.kill()
            except:
```

```
            pass
        #清空系统剪贴板
        ctypes.windll.user32.OpenClipboard(None)
        ctypes.windll.user32.EmptyClipboard()
        ctypes.windll.user32.CloseClipboard()
        time.sleep(1)

def start():
    jinyong.set(1)
    t=threading.Thread(target=funcJinyong)
    t.start()

buttonStart=tkinter.Button(root,text='开始考试',command=start)
buttonStart.place(x=20,y=10,width=100,height=20)

def stop():
    jinyong.set(0)
buttonStop=tkinter.Button(root,text='结束考试',command=stop)
buttonStop.place(x=130,y=10,width=100,height=20)

#模拟用,开启考试模式以后,所有内容都不再允许复制
entryMessage=tkinter.Entry(root)
entryMessage.place(x=10,y=40,width=230,height=20)

root.mainloop()
```

附录 I　软件分析与逆向工程

在软件分析和逆向工程领域,有大量的成熟工具以及针对不同工具和目的开发的各种插件,例如 IDA Pro、OllyDbg、WinDbg、W32DASM、PEid、ssdeep、DiStorm、DisView、LordPE、PIN、Universal PE Unpacker、Sample Chart Builder 等,可以说是数不胜数。下面简单列出使用 Python 开发或可以使用 Python 进行二次开发的工具和插件。

(1) PyEmu:可编写脚本的模拟器,对恶意软件分析非常有用。

(2) Immunity Debugger:著名的调试器,是在 OllyDbg 的源代码基础上建立起来的,外观和用法都与 OllyDbg 非常相似,并且两者共享很多的底层功能和控制。Immunity Debugger 带有内置的 Python 接口和专门用于研究漏洞和执行恶意软件分析的强大 API,是可编写脚本的 GUI 和命令行软件调试器,支持 exploit 编写、二进制可执行文件逆向工程等各种应用。

(3) Paimei:完全使用 Python 编写,是非常成熟的逆向工程框架,包括 PyDBG、PIDA、pGRAPH 等多个可扩展模块,可以执行大量静态分析和动态分析,例如模糊测试、代码覆盖率跟踪、数据流跟踪等。

(4) ropper:比较成熟的 ROP Gadgets 查找与可执行文件分析工具,其反汇编部分使



用了成熟的 Capstone 框架。

（5）WinAppDbg：纯 Python 调试器，没有本机代码，使用 ctypes 封装了许多与调试器有关的 Win32 API 调用，并且为操作线程和进程提供了强有力的抽象。利用该工具可以将自己编写的脚本附加为调试器、跟踪执行、拦截 API 调用，以及在待调试进程中处理事件，并且可以设置各种断点。

（6）YARA：恶意软件识别和分类引擎也可以利用 YARA 创建规则以检测字符串、入侵序列、正则表达式、字节模式等。既可以使用命令行模式下的 yara 工具扫描文件，也可以利用 YARA 提供的 API 函数将 yara 扫描引擎集成到 C 或 Python 语言编写的工具中。

（7）pefile：可以读取和处理 PE 文件。

（8）IDAPython：IDA 插件，IDAPython 是运行于交互式反汇编器 IDA 的插件，用于实现 IDA 的 Python 编程接口。IDA 在逆向工程领域具有广泛的应用，尤其是二进制文件静态分析，其强大的反汇编功能一直处于业内领先水平。IDAPython 插件使得 Python 脚本程序能够在 IDA 中运行并实现自定义的软件分析功能，通过该插件运行的 Python 脚本程序可以访问整个 IDA 数据库，并且可以方便地调用所有 IDC 函数和使用所有已安装的 Python 模块中的功能。目前 IDAPython 还不支持 Python3，较高版本的 IDA 中集成了 IDAPython 插件，如果需要安装或升级，需要登录其官方网站下载安装适合已安装 Python 和 IDA 版本的 IDAPython 插件。

（9）Hex-Rays Decompiler：IDA 插件，非常成熟的反编译插件。

（10）PatchDiff2：IDA 插件，主要用于补丁对比。

（11）BinDiff：IDA 插件，主要用于二进制文件差异比较。

（12）hidedebug：Immunity Debugger 插件，可以隐藏调试器的存在，用来对抗某些通用的反调试技术。

（13）IDAStealth：IDA 插件，可隐藏 IDA debugger 的存在，用来对抗某些通用的反调试技术。

（14）MyNav：IDA 插件，能够帮助逆向工程师完成一些最典型的任务，例如发现一些特定功能或任务是由哪些函数实现的，找出补丁前后函数的不同之处和数据入口。

（15）Lobotomy：一款应用于 Python 的安卓渗透测试工具包，可以帮助安全研究人员评估不同 Android 逆向工程任务。

特别需要注意的是，应尽量避免直接在本地物理主机上分析恶意软件，以免被恶意软件感染而造成不必要的损失。为了保证物理主机安全，同时也为了能够在分析环境被恶意软件感染之后快速恢复系统，建议使用 VirtualBox、VMware、QEMU 等虚拟机系统或沙箱系统进行保护。如果没有条件使用虚拟机或沙箱系统，最好使用 Deep Freeze、Truman、FPG 或其他类似软件来保护物理主机以防止系统被感染。

参 考 文 献

[1] Python 官方在线帮助文档. https://docs. python. org/3/.
[2] 董付国. Python 程序设计基础[M]. 北京: 清华大学出版社,2015.
[3] 董付国. Python 程序设计[M]. 北京: 清华大学出版社,2015.
[4] 董付国. Python 程序设计[M]. 2 版. 北京: 清华大学出版社,2016.
[5] 董付国. Python 可以这样学[M]. 北京: 清华大学出版社,2017.
[6] 张颖,赖勇浩,著. 编写高质量代码——改善 Python 程序的 91 个建议[M]. 北京: 机械工业出版社,2014.
[7] TJ. O' Connor. Python 绝技——运用 Python 成为顶级黑客[M]. 崔孝晨,武晓音,等译. 北京: 电子工业出版社,2016.